

**UNIVERSIDADE FEDERAL DE SERGIPE**  
**CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS**  
**PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

**Uma análise comparativa dos protocolos SNMP, Zabbix  
e MQTT, no contexto de aplicações de internet das coisas**

**Levi da Costa Mota**

**SÃO CRISTÓVÃO/SE**

2017

**UNIVERSIDADE FEDERAL DE SERGIPE**  
**CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS**  
**PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

**Levi da Costa Mota**

**Uma análise comparativa dos protocolos SNMP, Zabbix e MQTT, no contexto de aplicações de internet das coisas**

**Dissertação** apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como parte de requisito para obtenção do título de Mestre em Ciência da Computação.

**Orientador:** Prof. Dr. Edward David Moreno Ordoñez

**Coorientador:** Prof. Dr. Admilson de Ribamar Lima Ribeiro

**SÃO CRISTÓVÃO/SE**

2017

**FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL  
UNIVERSIDADE FEDERAL DE SERGIPE**

M917a Mota, Levi da Costa  
Uma análise comparativa dos protocolos SNMP, Zabbix e MQTT, no contexto de aplicações de internet das coisas / Levi da Costa Mota ; orientador Edward David Moreno Ordoñez. – São Cristóvão, 2017.  
128 f. : il.

Dissertação (mestrado em Ciências da computação) – Universidade Federal de Sergipe, 2017.

1. Internet - Programas de computador. 2. Simple Network Management Protocol (Protocolo de rede de computador). 3. Zabbix (Software). 4. Memória de acesso aleatório. 5. I. Ordoñez, Edward David Moreno, orient. II. Título.

CDU: 004.728.3.057.4

**Levi da Costa Mota**

# **Uma análise comparativa dos protocolos SNMP, Zabbix e MQTT, no contexto de aplicações de internet das coisas**

**Dissertação** apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como parte de requisito para obtenção do título de Mestre em Ciência da Computação.

## **BANCA EXAMINADORA**

Prof. Dr. Edward David Moreno Ordoñez, Orientador  
Universidade Federal de Sergipe (UFS)

Prof. Dr. Admilson de Ribamar Lima Ribeiro, Coorientador  
Universidade Federal de Sergipe (UFS)

Prof. Dr. Ricardo José Paiva de Britto Salgueiro  
Universidade Federal de Sergipe (UFS)

Prof. Dr. Adilson Eduardo Guelfi  
Universidade do Oeste Paulista (UNOESTE)

# **Uma análise comparativa dos protocolos SNMP, Zabbix e MQTT, no contexto de aplicações de internet das coisas**

Este exemplar corresponde à redação da Dissertação de Mestrado, sendo a defesa do aluno **Levi da Costa Mota** para ser aprovada pela Banca examinadora.

São Cristóvão - SE, 25 de agosto de 2017.

---

Prof. Dr. Edward David Moreno Ordoñez  
Orientador

---

Prof. Dr. Admilson de Ribamar Lima Ribeiro  
Coorientador

---

Prof. Dr. Ricardo Jose Paiva de Britto Salgueiro  
Membro

---

Prof. Dr. Adilson Eduardo Guelfi  
Membro

# Dedicatória

*À minha família, que me apoiou todo o tempo.*

# Agradecimentos

A Deus, Senhor de todas as coisas, por ter me sustentado durante a caminhada.

À minha família, principalmente a Helen, minha esposa, e aos meus filhos, Lavínia e Tito.

Ao meu orientador, professor Dr. Edward David Moreno Ordoñez, pela paciência e dedicação dispensada no processo de orientação, sendo peça chave no desenvolvimento deste trabalho, separando parte do seu escasso tempo para me orientar.

Ao professor Me. Marcos Dósea, pelas sugestões e opiniões que foram decisivas na elaboração da dissertação.

Ao professor Dr. Denysson Mota, pela disposição em ajudar, mesmo que à distância.

E a todas as pessoas que de alguma forma contribuíram para a realização deste trabalho.

Muito obrigado!

# Resumo

A complexidade e o crescimento das novas redes de objetos inteligentes vem gerando uma nova demanda pela manutenção desses dispositivos, com a necessidade de monitorar e controlar remotamente tais aparelhos sem consumir recursos significativos. Analisar o consumo de memória e o consumo de energia dos protocolos usados no gerenciamento dessas redes é uma maneira de evidenciar as melhores alternativas de protocolos para esse tipo de aplicação. Este trabalho realiza um estudo experimental, analisando o comportamento dos protocolos SNMP, Zabbix e MQTT, no tocante ao consumo de memória e ao consumo de energia elétrica, quando usados em uma aplicação de Internet das Coisas, com dispositivo sensor implementado sobre o ESP8266. O experimento é executado realizando o monitoramento de dispositivos em um ambiente com alguns Motes e um servidor Zabbix. O Mote construído coleta temperatura e umidade do ambiente e fornece, por meio de agente, informações de falha e desempenho para o servidor de gerenciamento. Um agente é implementado para cada protocolo examinado. O estudo analisa a memória ROM e a memória RAM ocupada pelo código do *firmware* resultante de cada agente, e acompanha a evolução do consumo de memória RAM ao longo do tempo. Também é feita a medição do consumo de energia de cada protocolo. Ao final, o estudo constata que os três protocolos analisados são suportados pela plataforma usada. O SNMP é o protocolo que consome menos memória do dispositivo, e o protocolo Zabbix é o que aloca mais memória ao longo do tempo. O trabalho também conclui que não há diferença significativa no consumo de energia entre os protocolos. Entretanto, o modelo de funcionamento do MQTT permite que o ESP8266 seja colocado em modo de suspensão nos momentos de inatividade, reduzindo em mais de 60% o consumo médio de energia do dispositivo.

**Palavras-chave:** Internet das Coisas; Objeto Inteligente; gerenciamento; protocolo; SNMP; Zabbix; MQTT; memória.



# Abstract

The complexity and the growth of new smart object networks has generated new demand for the maintenance of these devices, with the need to remotely monitor and control such equipment without consuming significant resources. Analyzing the memory consumption and the power consumption of the protocols used in the management of these networks is one way to highlight the best protocol alternatives for this type of application. This work performs an experimental study, analyzing the behavior of the SNMP, Zabbix and MQTT protocols, regarding memory consumption and power consumption, when used in an Internet of Things application, with sensor device implemented on the ESP8266. The experiment is executed by monitoring devices in an environment with some Motes and a Zabbix server. The Mote collects temperature and humidity from the environment and provides, through the agent, information about fault and performance to the management server. An agent is implemented for each protocol. The study analyzes the ROM memory and the RAM memory occupied by the firmware code resulting from each agent, and monitors the evolution of the RAM consumption over the time. The energy consumption of each protocol is also measured. In the end, the study finds that the three analyzed protocols are supported by the used platform. SNMP is the protocol that consumes less device memory, and the Zabbix protocol is what allocates more memory over the time. The study also concludes that there is no significant difference of energy consumption between the protocols. However, the operation model of MQTT allows the ESP8266 to be put into sleep mode during downtime, reducing the average device power consumption by more than 60%.

**Key-words:** Internet of Things; Smart Object; management; protocol; SNMP; Zabbix; MQTT; memory; energy.

# Lista de figuras

Figura 2.1 – Exemplos de sistemas embarcados.....	27
Figura 2.2 – Arquitetura de um Mote .....	29
Figura 2.3 – Arquitetura IoT .....	31
Figura 2.4 – Comparação entre as pilhas IP e 6LoWPAN. ....	33
Figura 2.5 – Arquitetura 6LoWPAN. ....	34
Figura 2.6 – Camadas existentes em roteadores de borda com suporte a 6LoWPAN.....	34
Figura 2.7 – Centro de Operações de Rede feito com Zabbix .....	36
Figura 2.8 – Arquitetura e componentes do Zabbix .....	38
Figura 2.9 – Arquitetura SNMP.....	39
Figura 2.10 – Parte da estrutura hierárquica da MIB.....	41
Figura 2.11 – Detalhes do protocolo Zabbix em modo passivo .....	44
Figura 2.12 – Arquitetura do MQTT .....	45
Figura 2.13 – Processo publish/subscribe usando pelo MQTT .....	45
Figura 2.14 – Aplicação mais simples do MQTT em IoT .....	47
Figura 2.15 – Cenário de uso do MQTT para gerenciamento em IoT.....	48
Figura 2.16 – Diagrama de blocos do ESP8266 .....	48
Figura 2.17 – Chip ESP8266EX .....	50
Figura 2.18 – Componentes da memória RAM do ESP8266. ....	51
Figura 2.19 – ESP-12.....	52
Figura 2.20 – Dimensões do ESP-12 .....	52
Figura 3.1 – Placa NodeMCU.....	60
Figura 3.2 – IDE Arduino usado para programar o ESP8266 .....	61
Figura 3.3 – Diagrama de funcionamento do mqttwarn .....	63
Figura 3.4 – Arduino Nano .....	64
Figura 3.5 – Funcionamento do medidor de corrente INA219 baseado em resistor shunt.....	65
Figura 3.6 – Sensor de corrente INA219 da Texas Instruments .....	66
Figura 3.7 – Esquemático do dispositivo medidor de energia .....	67
Figura 3.8 – Prototipação do dispositivo medidor de energia .....	67
Figura 3.9 – Disposição dos equipamentos usados na medição de energia.....	70
Figura 3.10 – Dispositivo medidor de energia em funcionamento .....	71
Figura 4.1 – Arquitetura de gerenciamento .....	72

Figura 4.2 – Diagrama de componentes do ambiente experimental .....	76
Figura 4.3 – Diagrama de dispositivos do ambiente experimental .....	76
Figura 4.4 – Ambiente experimental montado. ....	77
Figura 4.5 – Esquemático do dispositivo implementado .....	79
Figura 4.6 – Prototipação do dispositivo implementado .....	81
Figura 4.7 – Dispositivo implementado com NodeMCU / ESP8266 .....	82
Figura 4.8 – Diagrama do ambiente de desenvolvimento NodeMCU / ESP8266.....	82
Figura 4.9 – Página do Zabbix monitorando dispositivo com agente SNMP.....	88
Figura 4.10 – Página do Zabbix monitorando dispositivo com agente Zabbix. ....	90
Figura 4.11 – Publicações do agente MQTT. ....	92
Figura 4.12 – Página do Zabbix monitorando dispositivo com agente MQTT. ....	93
Figura 5.1 – Comparação quanto ao uso de memória ROM entre os agentes. ....	99
Figura 5.2 – Comparação quanto ao uso de memória RAM estática entre os agentes. ....	101
Figura 5.3 – Comparação da evolução do consumo de memória RAM pelos agentes.....	102
Figura 5.4 – Medições de potência elétrica dissipada.....	104
Figura 5.5 – Comparação da potência média dissipada. ....	105
Figura 5.6 – Comparação da energia total consumida. ....	105
Figura 5.7 – Comparação da previsão de duração da bateria. ....	105
Figura 5.8 – Corrente medida com agente MQTT e dispositivo entrando em suspensão. ...	106

# Lista de tabelas

Tabela 1.1 – Comparação entre os trabalhos relacionados. ....	23
Tabela 2.1 – Comparação entre os protocolos MQTT e HTTP. ....	47
Tabela 2.2 – Especificações do ESP8266. ....	49
Tabela 2.3 – Diferenças entre os modos de suspensão do ESP8266. ....	50
Tabela 3.1 – Comparação entre ferramentas de monitoramento. ....	57
Tabela 3.2 – Comparação entre dispositivos para IoT. ....	59
Tabela 3.3 – Especificações do sensor de corrente INA219. ....	65
Tabela 4.1 – Lista de componentes do dispositivo implementado. ....	80
Tabela 4.2 – Lista de OIDs implementados no agente SNMP. ....	87
Tabela 4.3 – Chaves implementadas no agente de monitoramento. ....	89
Tabela 4.4 – Tópicos implementados no agente de MQTT. ....	92
Tabela 5.1 – Uso de memória ROM e RAM dos agentes (em bytes). ....	98
Tabela 5.2 – Uso de memória ROM dos agentes (em bytes). ....	98
Tabela 5.3 – Uso de memória RAM dos agentes (em bytes). ....	100
Tabela 5.4 – Consumo de energia dos agentes. ....	104

# Lista de abreviaturas e siglas

A/D – *Analogue to Digital*

API – *Application Program Interface*

CCITT – *Comité Consultatif International Téléphonique et Télégraphique*

CERN – *European Organization for Nuclear Research*

CoAP – *Constrained Application Protocol*

CPU – *Central Processing Unit*

GPIO – *General Purpose Input/Output*

HTTP – *Hypertext Transfer Protocol*

I2C – *Inter-Integrated Circuit*

IETF – *Internet Engineering Task Force*

IoT – *Internet of Things*

IP – *Internet Protocol*

IPMI – *Intelligent Platform Management Interface*

ISO – *International Organization for Standardization*

LLD – *Low Level Discovery*

LoWPAN – *Low-power Wireless Personal Area Network*

MCU – *Microcontroller Unit*

MIB – *Management Information Base*

MQTT – *Message Queue Telemetry Transport*

NOC – *Network Operation Center*

OID – *Object Identification*

PCB – *Printed Circuit Board*

PDU – *Protocol Data Unit*

PWM – *Pulse-Width Modulation*

QoS – *Quality of Service*

RAM – *Random Access Memory*

ROM – *Read Only Memory*

RSE – *Redes de Sistemas Embarcados*

RSSF – *Redes de Sensores Sem Fio*

RTC – *Real Time Clock*

SDK – *Software Development Kit*

SNMP – *Simple Network Managament Protocol*

SoC – *System On a Chip*

TCP – *Transmission Control Protocol*

UDP – *User Datagram Protocol*

URI – *Universal Resource Identifier*

WSN – *Wireless Sensor Network*

XMPP – *Extensible Messaging and Presence Protocol*

# Sumário

1. Introdução.....	17
1.1 Descrição do problema.....	18
1.2 Questão de pesquisa .....	19
1.3 Objetivos .....	19
1.3.1 Objetivos Específicos .....	19
1.4 Trabalhos relacionados.....	20
1.5 Método de pesquisa.....	24
1.6 Contribuições .....	24
1.7 Organização do trabalho .....	25
2. Fundamentação Teórica.....	27
2.1 Sistemas Embarcados .....	27
2.2 Redes de Sistemas Embarcados .....	28
2.3 Redes de sensores sem fio.....	28
2.3.1 Mote.....	29
2.4 Computação Ubíqua.....	30
2.5 Internet das Coisas e objetos inteligentes.....	30
2.6 6LoWPAN.....	32
2.7 CoAP .....	34
2.8 Gerenciamento de redes de sensores sem fio.....	35
2.9 Zabbix .....	36
2.10 Protocolos usados no gerenciamento .....	39
2.10.1 Protocolo SNMP.....	39
2.10.2 Protocolo Zabbix .....	42
2.10.3 Protocolo MQTT .....	44
2.11 Microcontrolador ESP8266.....	48
2.11.1 Estrutura da memória do ESP8266.....	50
2.11.2 ESP-12 .....	51
3. Materiais e método de pesquisa.....	53
3.1 Método de pesquisa.....	53
3.1.1 Revisão da literatura .....	54
3.1.2 Seleção da plataforma de hardware .....	54
3.1.3 Escolha dos protocolos .....	54
3.1.4 Planejamento do experimento.....	54

3.1.5	Desenvolvimento do dispositivo sensor .....	54
3.1.6	Desenvolvimento do firmware do dispositivo sensor.....	55
3.1.7	Desenvolvimento dos agentes de gerenciamento .....	55
3.1.8	Desenvolvimento do dispositivo medidor de energia.....	55
3.1.9	Realização do experimento.....	55
3.1.10	Análise e interpretação dos dados coletados.....	56
3.2	Instrumentação .....	56
3.2.1	Zabbix .....	56
3.2.2	Escolha do ESP8266.....	58
3.2.3	Placa NodeMCU.....	58
3.2.4	Arduino IDE .....	60
3.2.5	Mosquitto.....	62
3.2.6	Mqttwarn .....	62
3.2.7	Arduino Nano .....	63
3.2.8	Sensor de Corrente INA219.....	64
3.3	Implementação do Dispositivo Medidor de Energia.....	66
3.4	Métricas usadas .....	67
3.4.1	Consumo de memória .....	67
3.4.2	Consumo de energia .....	69
4.	Desenvolvimento do ambiente experimental .....	72
4.1	Arquitetura de gerenciamento .....	72
4.1.1	Dispositivo Gerenciador .....	73
4.1.2	Dispositivo Gerenciado .....	74
4.1.3	Servidor e Console de Gerenciamento.....	74
4.1.4	Protocolo de Gerenciamento.....	74
4.1.5	Agente de Gerenciamento.....	75
4.2	Ambiente experimental .....	75
4.3	Instalação do Zabbix .....	77
4.4	Implementação do Dispositivo Gerenciado .....	78
4.5	Programação da placa .....	82
4.6	Implementação do firmware principal .....	84
4.6.1	Biblioteca Uptime.....	85
4.7	Implementação dos agentes e seus protocolos .....	86
4.7.1	Agente e protocolo SNMP.....	86
4.7.2	Agente e protocolo Zabbix .....	88
4.7.3	Agente e protocolo MQTT .....	91
4.8	Considerações finais do capítulo.....	94
5.	Resultados.....	96
5.1	Cenário do experimento .....	96



5.2	Consumo geral de memória .....	97
5.3	Memória ROM .....	98
5.4	Memória RAM inicial (estática) .....	100
5.5	Evolução do consumo total de memória RAM .....	101
5.6	Consumo de energia elétrica .....	103
6.	Conclusões.....	107
6.1	Trabalhos futuros .....	109
	Referências.....	111
	Apêndices.....	115
	APÊNDICE A Código fonte dos dispositivos.....	116
A.1	Fontes do firmware e dos agentes implementados para o dispositivo sensor .....	116
A.2	Fontes das bibliotecas implementadas para o dispositivo sensor .....	116
A.3	Fontes do dispositivo medidor de energia .....	116
	APÊNDICE B Configurações feitas no mqttwarn .....	117
B.1	Arquivo mqttwarn.ini .....	117
B.2	Arquivo samplefuncs.py .....	118
	APÊNDICE C Planilhas com as coletas de memória RAM em execução .....	119
C.1	Agente SNMP.....	119
C.2	Agente Zabbix .....	120
C.3	Agente MQTT .....	121
	APÊNDICE D Gráficos das grandezas elétricas medidas para cada protocolo .....	122
D.1	Agente SNMP.....	122
D.2	Agente Zabbix .....	123
D.3	Agente MQTT .....	124
D.4	Agente MQTT funcionando em modo de suspensão.....	125
	Anexos .....	126
	ANEXO A Fichas informativas (datasheets) de placas e componentes .....	127
A.1	Datasheets de componentes usados no dispositivo sensor.....	127
A.2	Datasheets de componentes usados no dispositivo medidor de energia .....	127
	ANEXO B Bibliotecas de terceiros usadas no experimento.....	128

# 1. Introdução

---

As redes de sistemas embarcados estão cada vez mais presentes no mundo atual, seja por meio de Redes de Sensores Sem Fio (RSSF), ou por causa dos novos objetos inteligentes que, ao serem interconectados e em seguida conectados à rede mundial, vêm criando o novo conceito da Internet das Coisas – *Internet Of Things* (IoT). Estima-se que atualmente cada pessoa tem ao menos dois objetos conectados à Internet, e que em 2020 esse número pode chegar a 20,8 bilhões (GEORGESCU; HUCANU, 2016) no mundo todo.

Esses objetos inteligentes são dispositivos sensores com limitações de processamento, alcance de rede, capacidade da bateria e quantidade de memória. Eles são geralmente empregados em aplicações com pouca ou nenhuma interação humana e suas redes geralmente são de larga escala, da ordem de milhares de nós (VASSEUR; DUNKELS, 2010). Para manter essas redes em bom funcionamento é essencial aplicar técnicas de gerenciamento que permitam monitorar o funcionamento dos elementos ou enviar comandos para alterar o seu comportamento (SHENG, Z. *et al.*, 2015).

Diversas pesquisas foram realizadas para tratar do gerenciamento de redes de dispositivos com recursos limitados, seja no contexto mais amplo do gerenciamento (SHENG, ZHENG GUO *et al.*, 2015), como também especificamente sobre o monitoramento dos elementos dessas redes (PAVENTHAN *et al.*, 2013). Várias abordagens foram usadas nessas pesquisas, como o uso de protocolos de gerenciamento consolidados como SNMP (WU; ZHU; DENG, 2008), o encapsulamento desses protocolos em camadas mais leves como 6LoWPAN (MUKHTAR *et al.*, 2008), o uso de dispositivos adicionais conectados aos elementos gerenciados criando uma interface de gerenciamento de *hardware* (LU *et al.*, 2015), o uso de *gateways* como tradutores do gerenciamento (SHENG, ZHENG GUO *et al.*, 2015) ou a implementação das funções desejadas diretamente nos elementos gerenciados por meio de agentes (CAI *et al.*, 2014).

Entretanto, não há estudo que faça comparação entre protocolos que podem ser usados para o gerenciamento de rede, analisando o consumo de memória e de energia dos agentes, a fim de garantir aplicabilidade destes protocolos no gerenciamento de dispositivos com poucos recursos. Diante desse cenário, surge a necessidade de saber o real impacto dos protocolos de gerenciamento de redes aplicados ao ambiente de internet das coisas, levando as características únicas desse ambiente.

O ESP8266 (ESPRESSIF SYSTEMS, 2017) vem se destacando como uma alternativa de baixo custo para implementação de objetos inteligentes. As características desse novo *System On a Chip* (SOC), como processador de 32 bits, memória RAM da ordem de 100KB e memória Flash de 512K, têm sido decisivas para o seu uso em diversas pesquisas científicas realizadas nos últimos dois anos (FERNANDES *et al.*, 2017; KODALI; SORATKAL, 2016; MARQUES; PITARMA, 2016; OLIVEIRA, 2017; SANTOS *et al.*, 2016).

Por outro lado, o Zabbix é uma ferramenta de código aberto capaz de monitorar a disponibilidade e o desempenho da infraestrutura de uma rede (HORST; PIRES; DÉO, 2015). A comunidade que desenvolve o produto também fez uma especificação simplificada, própria e aberta de um protocolo de gerenciamento, também chamado de Zabbix. Isso permite que um dispositivo que não suporte protocolos comuns como o SNMP possa incluir um agente Zabbix e expor seus dados. Já o protocolo MQTT, criado para o transporte de informações de telemetria, usa o modelo *publish/subscribe* e é adequado para uso em aplicações com dispositivos que têm restrição de recursos (LAMPKIN *et al.*, 2012). Por essas características ele vem sendo, nos últimos três anos, preferido para uso em ambientes de Internet das Coisas.

Ante ao exposto, pretende-se realizar uma pesquisa experimental que permita avaliar o consumo de memória e de energia dos protocolos SNMP, Zabbix e MQTT quando aplicados no gerenciamento de dispositivos sensores implementados sobre a plataforma do ESP8266.

## 1.1 Descrição do problema

O gerenciamento de redes tipicamente é realizado com a inclusão de agentes de gerenciamento nos elementos da rede. O agente é um componente de *software* que implementa um protocolo de gerenciamento e tem a capacidade de interagir com um servidor para prover as funcionalidades de gerenciamento desejadas. Esse componente é

adicionado ao *software* já existente para compor o *firmware* que será instalado no dispositivo. Esse novo componente acrescenta novas capacidades ao dispositivo, mas também adiciona consumo de recursos do equipamento que será gerenciado.

O *firmware* consome memória ROM e memória RAM do dispositivo em que é instalado. Depois de compilado, o código é armazenado na memória ROM do dispositivo. Para ser executado, o código é carregado para a memória RAM, a qual também é usada para armazenamento de dados estáticos e dados alocados dinamicamente. Assim, quanto menos espaço de código e de memória for ocupado pelo agente, mais espaço sobrá para o código do *software* principal.

É muito comum dispositivos usados em Internet das Coisas serem alimentados por baterias e funcionarem em campo sem alimentação por rede elétrica. A maneira como o *firmware* é implementado e executado pode influenciar na demanda por processamento, e consequentemente, no consumo de energia elétrica. Quanto mais eficiente for a implementação, menos energia será demandada do dispositivo.

Portanto, faz-se necessário saber qual o consumo de recursos que é acrescentado ao dispositivo com a inclusão do agente de gerenciamento que implementa o protocolo de comunicação com o servidor de gerenciamento.

## **1.2 Questão de pesquisa**

Qual o consumo de memória e de energia elétrica dos principais protocolos de gerenciamento de rede implementados em agentes instalados em um dispositivo ESP8266, quando aplicados em um ambiente de Internet das Coisas?

## **1.3 Objetivos**

O objetivo principal deste trabalho é analisar os protocolos de gerenciamento de redes SNMP, Zabbix e MQTT, com o propósito de avaliar o consumo de memória e o consumo de energia do agente de gerenciamento, do ponto de vista do desenvolvedor, no contexto de aplicações de Internet das Coisas, em sistemas embarcados implementados com o ESP8266.

### **1.3.1 Objetivos Específicos**

Para alcançar o objetivo principal da pesquisa, alguns objetivos específicos são

perseguidos, a saber:

- 1) Construir um dispositivo sobre o ESP8266 que possa ser usado em um ambiente de Internet das Coisas para realização dos experimentos. O dispositivo deve ser capaz de coletar informações do ambiente por meio de sensores. Ele também deve prover informações de falha e desempenho que permitam o monitoramento remoto. Tanto as informações do ambiente, quanto as informações de falha e desempenho, devem ser expostas pelos agentes de gerenciamento. A seção 5.1, que detalha o cenário do experimento, relaciona esse conjunto de informações;
- 2) Implementar um agente de gerenciamento que suporte os protocolos SNMP, Zabbix e MQTT, e que possa ser executado no *firmware* do dispositivo;
- 3) Desenvolver dispositivo que seja capaz de medir o consumo de energia elétrica de outro dispositivo alimentado por corrente contínua em execução;
- 4) Planejar e executar um estudo experimental que permita avaliar o consumo de memória e o consumo de energia dos agentes implementados, aplicados em um ambiente de gerenciamento de rede;
- 5) Analisar os resultados obtidos do experimento referente à aplicação de cada protocolo de gerenciamento, avaliando o uso de memória e a viabilidade de utilização do protocolo no ambiente estudado.

## 1.4 Trabalhos relacionados

Esta seção traz alguns trabalhos encontrados na área sobre gerenciamento e monitoramento de redes de sistemas embarcados e internet das coisas, incluindo a análise de protocolos usados e sugeridos pelos autores.

Em “Design and Implementation of Embedded SNMP Network Management Manager in Web-Based Mode” (WU; ZHU; DENG, 2008), os autores apontaram a significância do estudo em gerenciamento de redes de dispositivos embarcados, sendo o uso do protocolo SNMP uma alternativa interessante para esse gerenciamento. Eles apresentam uma arquitetura de gerenciamento de redes de sistemas embarcados baseada em SNMP e em modo *Web* em Linux. Ele reforça o grande número de funções disponíveis quando se realiza

o gerenciamento de redes com SNMP. Na arquitetura proposta pelo autor, o gerenciador SNMP foi implementado como um *Applet* Java e foi armazenado em um servidor *Web* embarcado no dispositivo. Tal arquitetura exige a instalação e a execução de componentes de *software* pesados no lado do dispositivo, incluindo um servidor *Web*, um servidor de SNMP, um agente SNMP e alguns módulos de comunicação e transporte, todos implementados em Java e executados em uma Máquina Virtual Java – *Java Virtual Machine* (JVM), sobre um sistema operacional Linux. Tal configuração exige um dispositivo com recursos bem acima da média dos dispositivos embarcados comumente encontrados no mercado. Por isso, o trabalho usa em seus experimentos a placa chinesa UP-TECH2410s, que tem processador ARM9 e executa uma versão embarcada do Linux. Além disso, não são feitas análises de métricas para avaliar o desempenho do protocolo no ambiente proposto, nem são analisados outros protocolos.

Mukhtar *et al.*, em “LNMP- Management Architecture for IPv6 Based Low-Power Wireless Personal Area Networks (6LoWPAN)” (MUKHTAR *et al.*, 2008), propõem o LoWPAN *Network Management Protocol* (LNMP) como a arquitetura de gerenciamento para WSN baseada em 6LoWPAN, mas com uma camada de filtro. Os autores argumentam que o SNMP não é suportado em redes 6LoWPAN por conta de limitações dos protocolos. É sugerido o uso de um *gateway* separando as redes IPv6 e 6LoWPAN. Na proposta, o SNMP é suportado apenas no lado IPv6, sendo traduzido pelo *gateway* para um formato reduzido suportado pela rede do lado 6LoWPAN. Os autores propõem definições MIB para a camada de adaptação 6LoWPAN, que cobrem aspectos como a classificação do dispositivo, a organização da rede e atributos de *broadcast* da rede 6LoWPAN. A análise feita pelos autores é sobre a latência das requisições em função do número de saltos realizados pelos pacotes para atingir o destino. Eles não avaliam o consumo de memória e não fazem comparação com outros protocolos.

Kuryla e Schönwälder (2011) fizeram um estudo para verificar se o protocolo SNMP poderia ser aplicado a dispositivos com recursos limitados. Eles implementaram um agente SNMP no sistema operacional Contiki e analisaram seu comportamento em uma plataforma AVR Raven de 8 bits. A análise foi feita em relação à memória ROM e memória RAM usada pelo agente. O método de análise de Kuryla foi o mesmo método deste trabalho. Porém, Kuryla analisou apenas o SNMP, não fazendo uma comparação com outros protocolos.

Os autores de “Management of Resource Constrained Devices in the Internet of

Things” (SEHGAL *et al.*, 2012) fazem uma avaliação de dois protocolos de gerenciamento aplicados ao ambiente de internet das coisas. O dispositivo usado é um AVR Raven. Os autores avaliam o protocolo SNMP aplicado ao monitoramento e o protocolo NETCONF aplicado à configuração do dispositivo. As métricas usadas também são baseadas no consumo de memória ROM e memória RAM. Entretanto, apesar da métrica usada ser a mesma desta pesquisa, os autores analisam apenas um protocolo em cada aspecto do gerenciamento.

“Design and Implementation of a WiFi Sensor Device Management System” é uma proposta de projeto de um sistema de gerenciamento de dispositivos sensores baseados em redes 802.11 (CAI *et al.*, 2014). No projeto apresentado, as funções de gerenciamento ficam separadas dos serviços IoT e são centralizadas numa entidade de gerenciamento. Essa entidade realiza o processo de gerenciamento usando o protocolo CoAP. Os autores propuseram um modelo de gerenciamento de dispositivos sensores baseado em protocolos leves como CoAP e que não exigem um *gateway* para se conectarem à Internet, permitindo assim o gerenciamento fim-a-fim desses dispositivos. É feito o uso de RESTful nas interfaces de comunicação entre os elementos, com o envio de requisições CoAP tipo Get, Put e Post. É usado um dispositivo CC3000 e o sistema operacional Contiki OS. É feita uma avaliação do uso de memória pelo agente, analisando as camadas do *firmware* separadamente. Também são analisados os tamanhos dos pacotes para cada tipo de mensagem.

No trabalho “A Perl-based SNMP Agent of Networked Embedded Devices for Smart-Living Applications” (LU *et al.*, 2015), Lu *et al.* propõem um agente SNMP para Microcontroladores – *Microcontroller Unit* (MCU) com o objetivo de controlar redes de sistemas embarcados, provendo uma maneira eficiente de interpretar o comando SNMP *get* em protocolo de comunicação de porta serial (RS-232). O agente proposto é implementado em linguagem Perl e não faz uso de MIB, para ter um tamanho reduzido. A proposta dos autores é separar o agente em outro dispositivo que roda Linux e que traduz toda solicitação SNMP em comandos que são transmitidos ao dispositivo embarcado por meio de comunicação serial. O agente é desenvolvido em linguagem Perl. A análise realizada é sobre o esforço de codificação. Os autores comparam o esforço de desenvolvimento do agente em Perl para um agente desenvolvimento em linguagem C, apontando que a linguagem sugerida traz redução no tempo de codificação.

Os autores de “Lightweight Management of Resource-Constrained Sensor Devices in

Internet of Things” (SHENG, ZHENG GUO *et al.*, 2015) propõem um modelo de gerenciamento para dispositivos com recursos limitados usando uma abordagem leve baseada em *Web Service* (WS) RESTful. Na proposta, são definidas funções de gerenciamento de dispositivos e feito um mapeamento dessas funções com métodos CoAP, que é o principal protocolo 6LoWPAN da camada de aplicação. Foi proposto um framework de gerenciamento onde o elemento gerenciável implementa a pilha 6LoWPAN com as funções de gerenciamento em protocolo CoAP e um roteador de borda da rede de sensores sem fio faz a conversão das requisições HTTP de gerenciamento em mensagens CoAP tratadas pelo elemento gerenciável. São avaliadas algumas métricas de desempenho: latência do protocolo, tamanho do pacote e perda de pacotes.

Um resumo mostrando as características dos experimentos realizados em cada trabalho, os protocolos analisados e as métricas aplicadas, é mostrado na Tabela 1.1.

Tabela 1.1 – Comparação entre os trabalhos relacionados.

Autor	Ano	Características				Protocolos analisados				Métricas		
		Dispositivo leve	Gerenciamento de ponta a ponta	Pilha 6LoWPAN	Pilha TCP/IP	SNMP	Zabbix	MQTT	Outro protocolo	Memória	Energia	Rede
WU; ZHU; DENG	2008		✓		✓	✓						
MUKHTAR <i>et al.</i>	2008	✓		✓					✓			
KURYLA e SCHÖNWÄLDER	2011	✓	✓	✓		✓				✓		
SEHGAL <i>et al.</i>	2012	✓		✓		✓			✓	✓		
CAI <i>et al.</i>	2014	✓	✓	✓					✓	✓		✓
LU <i>et al.</i>	2015				✓	✓						
SHENG <i>et al.</i>	2015	✓		✓					✓			✓
Este trabalho	2017	✓	✓		✓	✓	✓	✓		✓	✓	

Fonte: autoria própria

As colunas da Tabela 1.1 correspondentes às características do experimento de cada trabalho são detalhadas a seguir:

- **Dispositivo leve:** indica se o dispositivo usado no experimento possui recursos limitados e baixo consumo de energia, que são características comuns em dispositivos usados no contexto de Internet das Coisas;
- **Gerenciamento de ponta a ponta:** indica se o gerenciamento é feito com a comunicação direta entre o servidor de gerenciamento e o dispositivo gerenciado. Quando isso não acontece, costuma existir um outro dispositivo para fazer a tradução ou o tratamento das informações enviadas ou recebidas pelo dispositivo gerenciado;



- **Pilha 6LoWPAN:** indica se a comunicação usada pelo agente do experimento foi feita sobre a pilha 6LoWPAN. Nesses casos, o padrão de comunicação usado é o IEEE 802.15.4.
- **Pilha TCP/IP:** indica se a comunicação usada pelo agente do experimento foi feita sobre a pilha TCP/IP. Em geral isso acontece quando a comunicação é feita sobre os padrões IEEE 802.11b/g/n.

## 1.5 Método de pesquisa

Este trabalho é classificado, quanto à sua abordagem, como uma pesquisa quantitativa, pois suas amostras e resultados podem ser numericamente quantificados (GERHARDT; SILVEIRA, 2009). Quanto à sua natureza, esta é uma pesquisa aplicada, pois tem objetivo de gerar conhecimentos para aplicação prática. Além disso, é um trabalho explicativo, quando classificamos quanto ao objetivo, dado que procura identificar os fatores que contribuem para a ocorrência de fenômenos.

A pesquisa científica é considerada experimental quando o pesquisador sistematicamente provoca alterações no ambiente pesquisado a fim de observar se as intervenções produzem algum resultado esperado. Portanto, em relação aos procedimentos usados, este trabalho classifica-se como uma pesquisa experimental, pois é realizado um estudo experimental com o objetivo de analisar os efeitos no consumo de memória do *firmware* do dispositivo, mediante alterações no agente de gerenciamento com a substituição dos protocolos usados pelo agente.

## 1.6 Contribuições

Este trabalho de pesquisa traz diversas contribuições, quais sejam:

- 1) Verificação e identificação dos protocolos, entre os analisados, que podem ser implementados em agentes executados na plataforma ESP8266;
- 2) Identificação dos protocolos, entre os analisados, cujos agentes consomem menos recursos dos dispositivos em que são executados, tanto em termos de memória quanto em relação à energia elétrica, e que mais se adequam ao ambiente de Internet das Coisas;
- 3) Melhoria no processo desenvolvimento de dispositivos para Internet das Coisas,

possibilitando a construção de objetos inteligentes gerenciáveis que, por esta característica, teriam sua vida útil estendida, reduzindo assim o custo desses objetos e suas redes;

- 4) Disponibilização de um modelo simplificado de circuito de dispositivo sensor com base no ESP8266 e que pode servir de base para novas pesquisas na área;
- 5) Disponibilização de código fonte de *firmware* para dispositivo sensor baseado no ESP8266;
- 6) Disponibilização de código fonte de agentes de gerenciamento que implementam os protocolos SNMP, Zabbix e MQTT na plataforma ESP8266. Os códigos podem ser acessados por meio da plataforma GitHub <sup>1</sup>, conforme detalhado no APÊNDICE A.

## 1.7 Organização do trabalho

No capítulo 1 é feita uma introdução sobre o assunto, levantando as questões iniciais que envolvem o tema do trabalho e as justificativas para o desenvolvimento desse tema. Também são analisados trabalhos relacionados ao tema proposto, identificando suas propostas e as análises feitas em cada trabalho. Nesse capítulo é apresentado o problema de pesquisa, o objetivo geral e os objetivos específicos. Também é abordado o método de pesquisa usado para a elaboração do trabalho e suas contribuições científicas.

O capítulo 2 apresenta a fundamentação teórica, fazendo um levantamento das várias tecnologias e conceitos que envolvem o tema e que foram necessários para o desenvolvimento do trabalho.

O capítulo 3 apresenta a metodologia usada neste trabalho de pesquisa. Neste capítulo o método usado é descrito, e as atividades realizadas são detalhadas. Também são mostrados todos os materiais e recursos usados para a realização do experimento. Ao final são detalhadas as métricas usadas para realizar a avaliação.

O capítulo 4 trata do experimento, detalhando tudo o que foi feito para o seu desenvolvimento. Aqui são detalhadas todas as implementações de *hardware* e de *software* necessários à construção do dispositivo e do *firmware* com seus agentes.

---

<sup>1</sup> Disponível em <https://github.com/levicm/nes-management/>

No capítulo 5 são feitas as análises dos dados coletados durante o experimento, apontando os resultados alcançados nesses testes. Aqui os dados são discutidos e gráficos são apresentados para melhor percepção e entendimento dos resultados obtidos.

Por fim, no capítulo 6 são apresentadas as conclusões tiradas do trabalho. Lá os objetivos definidos são confrontados para verificar se de fato foram alcançados. Ao final do capítulo, são sugeridos trabalhos futuros para continuidade do tema.

## 2. Fundamentação Teórica

---

Este capítulo aborda as ideias e os principais conceitos sobre o tema pesquisado, sintetizando os conceitos já publicados.

Aqui serão abordados, de forma encadeada, os conceitos de Sistemas Embarcados e suas redes, a evolução dessas redes para a Internet das Coisas e seus componentes, os Objetos Inteligentes, os fundamentos do gerenciamento desse tipo de rede assim como seus protocolos. Também são tratados os fundamentos dos componentes que são usados no experimento, como a ferramenta de gerenciamento Zabbix e o ESP8266.

### 2.1 Sistemas Embarcados

A base deste trabalho são os sistemas embarcados. De forma simplificada, qualquer computador que seja um componente de um sistema maior e que se baseia no seu próprio microprocessador, pode ser considerado um sistema embarcado (WOLF, 2002). Um sistema embarcado é um dispositivo que combina *hardware* e *software* usado para executar alguma função específica. A computação embarcada é atualmente uma disciplina de engenharia estabelecida com seus próprios princípios e conhecimentos (WOLF, 2002).

Figura 2.1 – Exemplos de sistemas embarcados



Fonte: autoria própria

Um sistema embarcado é composto por um microcontrolador, algum *hardware* e

*software*, e atua como um dispositivo para um determinado fim, possuindo geralmente custo baixo e alto desempenho (WU; ZHU; DENG, 2008). Eles estão sendo largamente usados em smartphones e diversos outros dispositivos eletrônicos do nosso cotidiano, como TVs, aparelhos de som, tocadores de Blu-ray, etc. Em um futuro muito próximo, os sistemas embarcados se tornarão essenciais para o desenvolvimento de casas inteligentes (LU *et al.*, 2015), indústrias inteligentes e até cidades inteligentes (ZANELLA *et al.*, 2014).

Em geral, vários desses dispositivos possuem recursos limitados, em termos de energia, capacidade computacional, memória, largura de banda, ou mesmo capacidades limitadas, como é o caso dos nós de Redes de Sensores Sem Fio (RSSF) – *Wireless Sensor Network* (WSN) (RANTOS *et al.*, 2012).

## **2.2 Redes de Sistemas Embarcados**

Redes de Sistemas Embarcados (RSE) – *Networked Embedded System* (NES) são essencialmente nós embarcados espacialmente distribuídos e interconectados por uma infraestrutura de comunicação com fio ou sem fio (ZURAWSKI, 2005).

A convergência da computação, comunicação e controle e os recentes avanços na miniaturização de dispositivos computacionais com capacidade de comunicação e com baixo consumo de energia tornou possível a construção de Redes de Sistemas Embarcados que podem ser profundamente envolvidas com o mundo físico e real, incluindo ambientes residenciais, automóveis, prédios e pessoas (DINI; SAVINO, 2010).

Tal evolução também fez surgir diversos outros tipos de NES, como as WSN, os ambientes residenciais inteligentes, os ambientes de vida assistida, as cidades inteligentes, e o conceito de computação ubíqua.

## **2.3 Redes de sensores sem fio**

Uma das primeiras especializações das Redes de Sistemas Embarcados foi a Rede de Sensores Sem Fio (RSSF) – *Wireless Sensor Network* (WSN). As RSSFs emergiram da ideia de que pequenos sensores poderiam ser usados para coletar informações de ambientes físicos em uma gama de situações, desde o rastreamento de incêndios florestais e a observação animal até o gerenciamento da agricultura e monitoramento industrial (VASSEUR; DUNKELS, 2010).

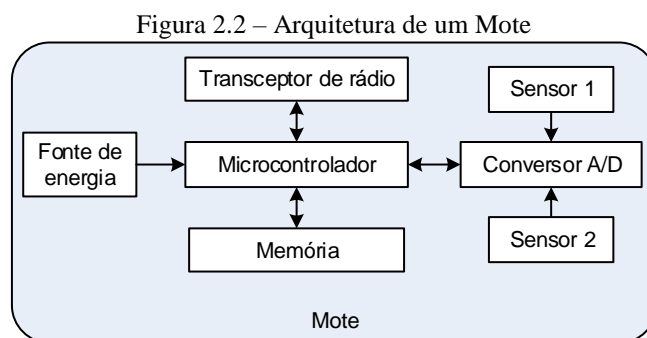
Nessas redes, os sistemas embarcados são dispositivos sensores, que transmitem por meio de comunicação sem fio as informações coletadas até uma estação central. Os sensores ajudam um ao outro no trabalho de transmitir a informação, aumentando assim o alcance da rede (VASSEUR; DUNKELS, 2010).

O campo de pesquisa das redes de sensores sem fio se tornou bastante ativo desde o ano 2000. Sua comunidade de pesquisa desenvolveu ao longo desse tempo importantes mecanismos, algoritmos, protocolos e abstrações. Os nós sensores dessas redes normalmente usam bateria como fonte de alimentação. Aumentar o tempo de vida significa reduzir o consumo de energia dos nós. Por causa disso, diversos mecanismos de redução de consumo foram projetados, estudados e avaliados tanto em ambientes simulados quanto em ambientes reais (VASSEUR; DUNKELS, 2010).

### 2.3.1 Mote

Os nós de uma RSSF também são conhecidos como Motes (DAGALE *et al.*, 2015; LEVIS *et al.*, 2005; WANG *et al.*, 2006). O termo Mote surgiu por volta do ano 2002 entre os pesquisadores da Universidade de Berkeley, nos Estados Unidos, após a apresentação da Plataforma Mica (HILL, J.; CULLER, 2001).

Para ser considerado um Mote, o dispositivo deve possuir alguns componentes como microcontrolador, memória, transceptor de rádio, fonte de energia (geralmente bateria), conversor analógico digital e alguns sensores. A arquitetura típica do Mote com esses componentes é mostrada na Figura 2.2



Fonte: autoria própria

A partir do ano 2002, várias soluções de Motes foram publicadas por universidades e um novo nicho de mercado foi criado. Resultados de pesquisas se transformaram em produtos comerciais, como a sequência de Motes Mica (CROSSBOW, 2002, 2005; CROSSBOW TECHNOLOGY, 2008; HILL, J. L.; CULLER, 2002) que passou a ser

comercializado pela Crossbow Technology. E até grandes corporações, como a Intel, entraram no mercado.

## 2.4 Computação Ubíqua

Computação ubíqua, também chamada de computação pervasiva, é um campo de estudo baseado no conceito de que os computadores se tornam imersos nos ambientes que nos envolvem, fazendo parte dos objetos. O estudo desse campo iniciou no final dos anos 1980, por Mark Wiser, que acreditava que a computação iria se integrar ao nosso ambiente diário (VASSEUR; DUNKELS, 2010).

O mundo atual está sendo cada vez mais preenchido por redes pervasivas de dispositivos ricos em sensores e por computação embarcada, que injetam computação no mundo físico de forma ubíqua e invisível (ESTRIN *et al.*, 2002).

Como qualquer tecnologia em ascensão, existem diversos desafios no desenvolvimento de redes de objetos inteligentes. Tanto desafios técnicos quanto desafios não técnicos. Entre os desafios técnicos pode-se destacar o desafio em relação à rede, mais especificamente o que diz respeito ao tamanho destas redes. Redes de objetos inteligentes são potencialmente de larga escala em termos de números de nós (VASSEUR; DUNKELS, 2010). Os impedimentos mais sérios para o avanço da computação pervasiva são os desafios técnicos ligados ao imenso número de elementos distribuídos e o acesso limitado a tais elementos (ESTRIN *et al.*, 2002).

## 2.5 Internet das Coisas e objetos inteligentes

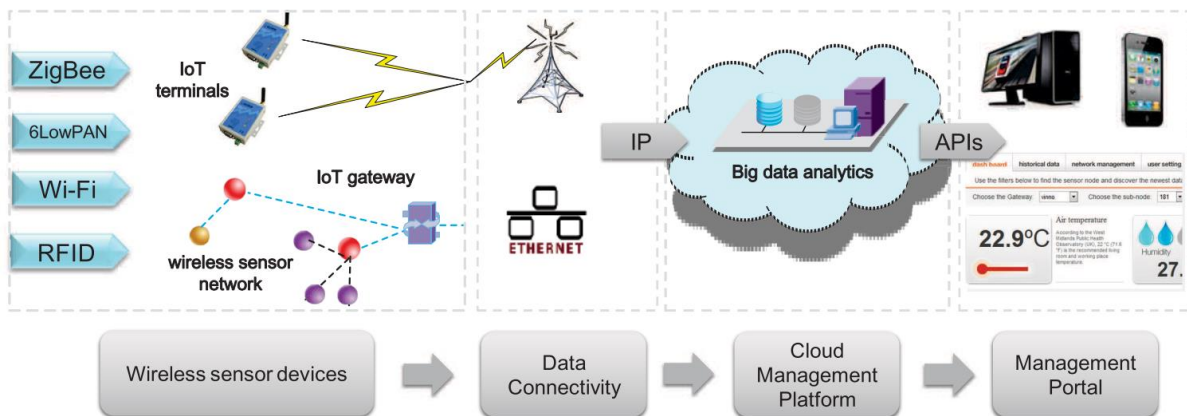
À medida que as redes de computadores foram evoluindo, mais notadamente por conta da Internet, uma nova revolução surgiu, a Internet das Coisas – *Internet of Things* (IoT), com a visão de que os dispositivos embarcados estão passando a possuir números IP e passando a fazer parte da Internet (SHELBY; BORMANN, 2009). Esses dispositivos embarcados passaram a ser chamados também de *Smart Objects* ou Objetos Inteligentes.

O novo paradigma de IoT expande o conceito de redes de sistemas embarcados a um novo patamar. Entender como os conceitos anteriores são tratados nessa nova arquitetura se faz necessário.

Uma definição formal para objeto inteligente é dada por um item equipado com

sensores e atuadores, um microcontrolador, um dispositivo de comunicação e uma fonte de força. Os sensores e atuadores dão ao objeto inteligente a capacidade de interagir com o mundo físico. O microcontrolador permite que o objeto inteligente processe e transforme os dados coletados. O dispositivo de comunicação dá ao objeto inteligente o poder de transmitir os dados para o mundo externo ou de receber dados de outros objetos inteligentes. E a fonte de força provê energia elétrica para que ele possa funcionar (VASSEUR; DUNKELS, 2010). O objeto inteligente pode ser considerado um tipo de sistema embarcado, sendo que a principal diferença entre os dois é que o objeto inteligente tem a capacidade de comunicação, que não é requerida em um sistema embarcado qualquer.

Figura 2.3 – Arquitetura IoT



Fonte: (SHENG, Z. *et al.*, 2015).

Após analisar a arquitetura proposta por várias organizações, Sheng (2015) detalha na Figura 2.3 a arquitetura da IoT, destacando os elementos: 1) dispositivos sensores, 2) conectividade de dados, 3) plataforma de gerenciamento em nuvem e 4) portal de gerenciamento (SHENG, Z. *et al.*, 2015).

- 1) Os dispositivos sensores formam o alicerce da IoT. Eles capturam informações de diferentes objetos e enviam para cima, através de *gateways*, por meio de redes sem fio, cabeadas ou mistas (SHENG, Z. *et al.*, 2015);
- 2) A conectividade de dados funciona como um gateway que traduz os dados capturados para um formato padrão e os envia para a plataforma de gerenciamento em nuvem (SHENG, Z. *et al.*, 2015);
- 3) A plataforma de gerenciamento em nuvem forma o núcleo da IoT, provendo um conjunto comum de operações como gerenciamento de dispositivos, conversão de protocolos e redirecionamento de rotas (SHENG, Z. *et al.*, 2015);



- 4) Portal de gerenciamento: ferramenta que permite visualizar os dados capturados, podendo contemplar as funções de gerenciamento dos objetos inteligentes.

Analisando a arquitetura proposta por Sheng, é possível observar que ele considera que o gerenciamento em Internet das Coisas compreende todo o tratamento das informações obtidas dos objetos inteligentes, sejam elas relacionadas à manutenção da rede, como também relacionadas aos dados finalísticos de sensoriamento.

## 2.6 6LoWPAN

O 6LoWPAN se tornou uma tendência em pesquisas acadêmicas e em produtos comerciais para o ambiente de IoT. Entretanto, o desenvolvimento de dispositivos modernos, como o ESP8266, que conseguem trabalhar diretamente com os padrões IEEE 802.11b/g/n de forma eficiente, pode tornar o uso do 6LoWPAN dispensável na construção de soluções IoT. Ainda assim, conhecer o funcionamento do 6LoWPAN é importante para entender as diferenças entre as soluções que são baseadas nessa pilha de protocolos e as que não são.

Até o início do ano 2000 apenas dispositivos mais robustos suportavam conexão com redes IP. Os protocolos de Internet tradicionais costumam demandar recursos, por conta de fatores relacionados a segurança, gerenciamento, tamanho dos quadros e as novas tecnologias de integração. Para garantir segurança é comum fazer uso de protocolos que permitam criptografia, autenticação e autorização, que usam técnicas complexas e que demandam processamento. Protocolos consolidados de gerenciamento dessas redes, como o SNMP, são frequentemente complexos e pesados. Protocolos de comunicação atuais, como oIPv6, exigem um tamanho de quadro de 1280 bytes. Está sendo cada vez mais comum lançar mão dos *Web Services* como tecnologia de integração entre sistemas, trazendo consigo padrões complexos e pesados, a exemplo do XML, SOAP, etc. (SHELBY; BORMANN, 2009).

Fazer os sistemas embarcados integrarem-se por meio de uma rede que usa protocolos de Internet poderia trazer inúmeros benefícios (SHELBY; BORMANN, 2009), tais como:

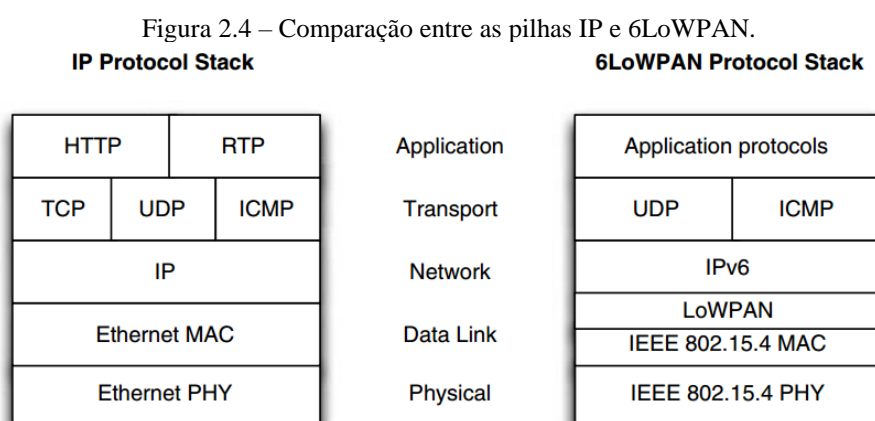
- Dispositivos IP poderiam ser conectados facilmente com outras redes IP;
- Novas redes IP poderiam aproveitar infraestruturas já existentes;

- Tecnologias baseadas em IP existem há décadas e já provaram ser escaláveis;
- A tecnologia IP é uma especificação aberta;
- Ferramentas para gerenciamento e diagnóstico de redes IP já existem e poderiam ser usadas.

O 6LoWPAN objetiva trazer essas vantagens às novas redes de pequenos dispositivos. Ele é um conjunto de padrões que possibilita o uso eficiente do IPv6 sobre dispositivos embarcados com poucos recursos de energia e de comunicação, através de uma camada de adaptação e, conseqüentemente, a otimização dos protocolos relacionados (SHELBY; BORMANN, 2009).

O IEEE 802.15.4 foi lançado em 2003 como o primeiro padrão global de rádio de baixa potência, chamado de *Low-power Wireless Personal Area Network* (LoWPAN) (SHELBY; BORMANN, 2009). Logo depois, a ZigBee Alliance desenvolveu uma solução de rede *ad-hoc* sobre o padrão IEEE 802.15.4 que se chamou ZigBee e se tornou muito popular. Apesar da popularidade, essas redes eram isoladas e tinham problemas para integração com a Internet.

Os padrões 6LoWPAN foram definidos sobre o padrão IEEE 802.15.4 e tornaram possível a integração de redes isoladas com a Internet por meio do uso do IPv6. A Figura 2.4 mostra a comparação entre as pilhas TCP/IP e 6LoWPAN.

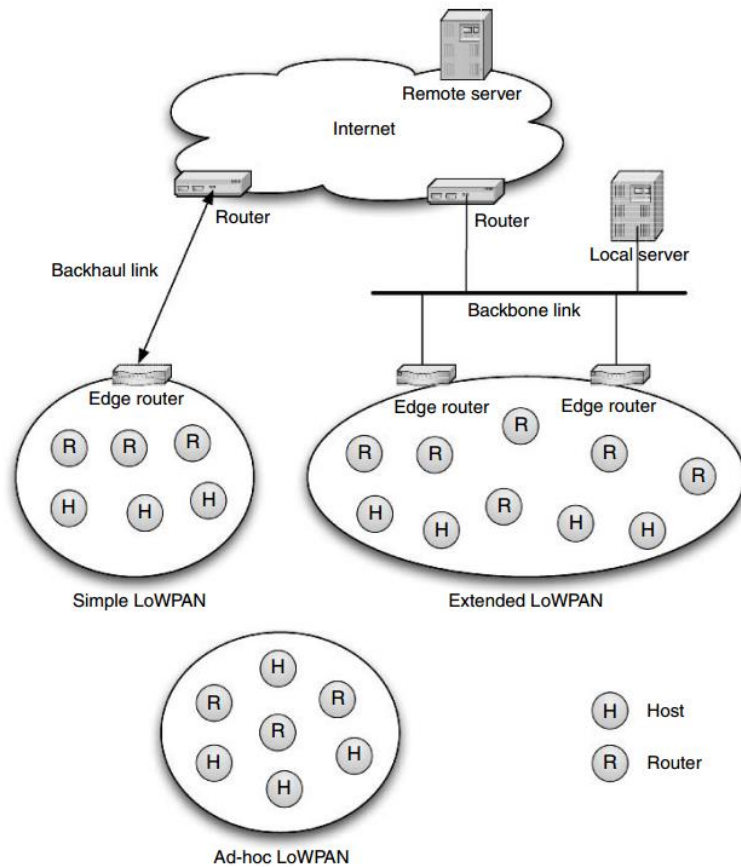


Fonte: (SHELBY; BORMANN, 2009).

A arquitetura 6LoWPAN funciona conectando ilhas de redes de dispositivos embarcados sem fio. Cada ilha é uma rede LoWPAN cujos pacotes IP podem entrar e sair, mas que não funcionam como passagem para outras redes. Os tipos dessas redes independentes são: LoWPAN simples, LoWPAN estendida e LoWPAN *ad-hoc* (SHELBY;

BORMANN, 2009). O modelo dessa arquitetura pode ser visto na Figura 2.5.

Figura 2.5 – Arquitetura 6LoWPAN.



Fonte:(SHELBY; BORMANN, 2009).

Figura 2.6 – Camadas existentes em roteadores de borda com suporte a 6LoWPAN.

IPv6	
Ethernet MAC	LoWPAN adaptation
	IEEE 802.15.4 MAC
Ethernet PHY	IEEE 802.15.4 PHY

Fonte:(SHELBY; BORMANN, 2009).

Uma LoWPAN é conectada à Internet por meio de um roteador de borda, conforme visto na Figura 2.5. Esse equipamento precisa conhecer as duas tecnologias e tratar o tráfego que entra e sai das LoWPANs para à Internet, portanto, precisa suportar parte das duas pilhas de protocolo (SHELBY; BORMANN, 2009), como demonstra a Figura 2.6.

## 2.7 CoAP

O *Constrained Application Protocol* (CoAP) é um protocolo *Web* genérico apresentado pela *Internet Engineering Task Force* (IETF) e projetado para atender a

requisitos de ambientes de redes com recursos escassos, como redes de sistemas embarcados (PAVENTHAN *et al.*, 2013).

O CoAP suporta comunicações confiáveis e não confiáveis baseado no paradigma requisição/resposta, adicionalmente suportando notificações assíncronas sobre o protocolo *User Datagram Protocol* (UDP) (PAVENTHAN *et al.*, 2013).

Além disso, o CoAP obedece ao padrão REST, abstraindo todos os objetos da rede como se fossem recursos. Cada recurso corresponde a um único Identificador Universal de Recurso – *Universal Resource Identifier* (URI) a partir do qual os recursos podem ser manipulados independente de estado, usando as operações GET, PUT, POST, DELETE, e assim por diante (SHENG, ZHENG GUO *et al.*, 2015).

O CoAP se tornou uma das alternativas mais utilizadas como protocolo de camada de aplicação para soluções 6LoWPAN. Boa parte das propostas e pesquisas acadêmicas que usam a pilha 6LoWPAN se apoiam ou são construídas sobre o protocolo CoAP, visto que ele vem sendo considerado o protocolo padrão da camada de aplicação dessa pilha (CAI *et al.*, 2014; PAVENTHAN *et al.*, 2013; SHENG, ZHENG GUO *et al.*, 2015; ZANELLA *et al.*, 2014). Isso gera ainda mais complexidade para as soluções propostas, que em geral se baseiam em encapsulamento ou conversão para o CoAP.

## 2.8 Gerenciamento de redes de sensores sem fio

Historicamente as redes computacionais sempre foram marcadas pela complexidade, diversidade e crescimento. Essas características dificultam a manutenção desses ambientes. Adicionalmente, como as redes de sistemas computacionais vêm se tornando críticas para os ambientes modernos de negócio, monitorar e garantir sua confiabilidade em desempenho é absolutamente requerido (ETIENNE, 2014). Uma forma de manter as redes de computadores em bom funcionamento é o uso de soluções que permitam o gerenciamento dos elementos dessas redes.

O gerenciamento de redes soma todas as ferramentas, procedimentos, métodos e atividades necessárias à operação, administração, manutenção e provisionamento de elementos de uma rede. A *International Organization for Standardization* (ISO) define os principais domínios do gerenciamento de redes como sendo: falha, configuração, contabilização, desempenho e segurança (MAGHETI; CIOBANU; POPOVICI, 2010).

Modelos e protocolos foram desenvolvidos para esse fim, e o principal expoente dessas tecnologias é o protocolo SNMP. O protocolo possui uma série de versões (SNMPv1, SNMPv2c, SNMPv3), uma linguagem para descrever modelos de dados (MIBs), e um conjunto de modelos padronizados (MAGHETI; CIOBANU; POPOVICI, 2010).

As redes de sistemas embarcados também trazem dificuldades similares às redes comuns. Além disso, os dispositivos inteligentes, que costumam ter diversas limitações de recursos, acrescentam ainda mais complexidade à tarefa de gerenciamento dessas redes. Para manter esses dispositivos sensores, por exemplo, monitorando seu desempenho ou enviando comandos ao nó sensor, é essencial que se use um protocolo de comunicação que seja eficiente e não consuma recursos consideráveis (SHENG, Z. *et al.*, 2015).

## 2.9 Zabbix

O Zabbix é uma ferramenta moderna, de código aberto e multiplataforma, com sistema de monitoramento distribuído, capaz de monitorar a disponibilidade e o desempenho da infraestrutura de uma rede, além de aplicações (HORST; PIRES; DÉO, 2015).

Figura 2.7 – Centro de Operações de Rede feito com Zabbix



Fonte: autoria própria.

Ele surgiu da iniciativa de Alexei Vladishev de criar uma ferramenta de monitoramento que, diferente do que havia no mercado, não fosse cara, não fosse de difícil manutenção, nem exigisse conhecimentos avançados para utilização. Hoje o Zabbix é uma das soluções de monitoramento mais populares de código aberto (HORST; PIRES; DÉO, 2015). A Figura 2.7 mostra um Centro de Operações de Rede – *Network Operation Center*

(NOC) de um órgão público do estado de Sergipe, feito com base no Zabbix.

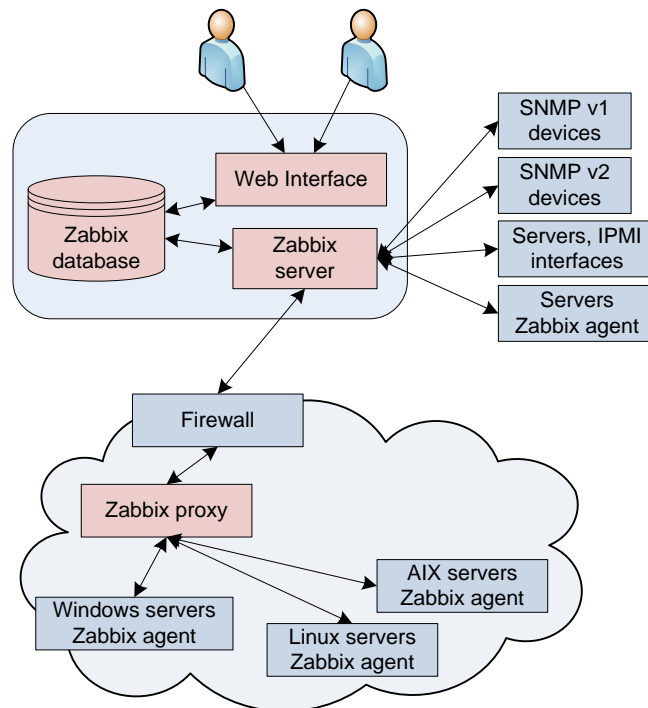
A ferramenta possui dezenas de módulos, mas as principais funcionalidades são listadas a seguir:

- Autodescoberta de dispositivos de rede;
- Autodescoberta de recursos do host;
- Descoberta de baixo nível – *Low Level Discovery* (LLD);
- Possibilidade de monitoramento distribuído com administração centralizada;
- Aplicação servidor compatível com ambiente GNU/Linux, IBM AIX, HP-UX, AIX, Solaris e família BSD;
- Tradução para vários idiomas;
- Autenticação;
- Auditoria;
- Suporte nativo a SNMP;
- Monitoramento via *Intelligent Platform Management Interface* (IPMI)
- Monitoramento de aplicações *Web*;
- Monitoramento de ambientes virtualizados;
- Envio de alertas via e-mail, SMS, *Extensible Messaging and Presence Protocol* (XMPP) e scripts personalizados.

#### **2.9.1.1 Arquitetura do Zabbix**

A arquitetura básica do Zabbix é mostrada na Figura 2.8. Normalmente o Zabbix é instalado em uma única máquina, mas pode haver razões para separar seus elementos. Uma instalação básica do Zabbix contém ao menos o Servidor Zabbix, a Interface *Web* do Zabbix e o Banco de Dados Zabbix. Mas outros componentes como o Agente Zabbix e o Proxy Zabbix também fazem parte de sua arquitetura.

Figura 2.8 – Arquitetura e componentes do Zabbix



Fonte: (HORST; PIRES; DÉO, 2015).

A seguir são detalhados os componentes da ferramenta:

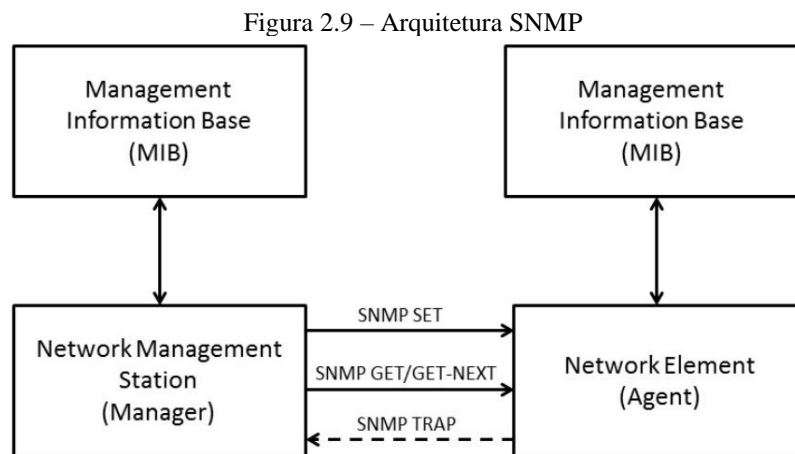
- **Servidor Zabbix:** é o componente central do sistema. É para ele que os agentes enviam as informações do equipamento que está sendo monitorado;
- **Banco de dados Zabbix:** é onde as informações de monitoramento dos dispositivos são armazenadas. É acessado pelo Servidor Zabbix e pela Interface Web;
- **Interface Web Zabbix:** é uma aplicação Web que permite acessar e visualizar as informações obtidas dos agentes;
- **Agente Zabbix:** porção de *software* que é executada no dispositivo gerenciado e que envia dados para o Servidor Zabbix. O agente acompanha ativamente o consumo de recursos do dispositivo gerenciado;
- **Proxy Zabbix:** parte opcional do Zabbix que permite distribuir a coleta dos dados de gerenciamento.

## 2.10 Protocolos usados no gerenciamento

### 2.10.1 Protocolo SNMP

*Simple Network Management Protocol* (SNMP) é um protocolo de gerenciamento e monitoramento de rede que roda sobre o protocolo UDP (PAVENTHAN *et al.*, 2013). Sua última especificação é a RFC-1157, da (CASE *et al.*, 1990).

O SNMP é um protocolo da camada de aplicação da pilha TCP/IP. Seu modelo arquitetural tem uma coleção de estações de gerenciamento de rede e de elementos de rede, os chamados dispositivos gerenciados. Cada dispositivo gerenciado contém um *software* chamado de agente, que é responsável por manipular as requisições do gerenciador (ETIENNE, 2014). Essa arquitetura é mostrada na Figura 2.9.



Fonte: (ETIENNE, 2014).

O SNMP é um protocolo que primordialmente permite o monitoramento de dispositivos. Mas adicionalmente, ele também é capaz de lidar com tarefas de configuração e modificar parâmetros remotamente (ETIENNE, 2014).

São quatro operações previstas pelo SNMP, conforme mostrado na Figura 2.9:

- 1) SNMP GET;
- 2) SNMP GET-NEXT;
- 3) SNMP SET;
- 4) SNMP TRAP.

Entretanto, existem 5 tipos de mensagens definidas, cada uma com sua especificação



de conteúdo definido em uma *Protocol Data Unit* (PDU):

- 1) GetRequest-PDU;
- 2) GetNextRequest-PDU;
- 3) Get-Response-PDU
- 4) SetRequest-PDU;
- 5) Trap-PDU.

O modelo de comunicação entre o gerenciador e o agente é do tipo requisição-resposta. Um gerenciador pode acessar informações do agente por meio de operações SNMP GET e SNMP GET-NEXT. O SNMP GET solicita um objeto específico. O SNMP GET-NEXT permite ir solicitando todos os objetos disponíveis no agente. O agente responde a essas operações enviando um GetResponse-PDU preenchido com o valor requisitado. Já a operação SNMP SET permite modificar um valor de um objeto. Ele é usado para alterar remotamente parâmetros usados pelo dispositivo (ETIENNE, 2014).

O SNMP foi originalmente projetado para ser um protocolo de monitoramento. Mas seus projetistas incluíram a operação SNMP TRAP. Essa operação habilita o agente a realizar notificações ao gerenciador sobre um evento específico. Assim, o SNMP TRAP é enviado sem a necessidade de uma requisição prévia (ETIENNE, 2014).

O protocolo SNMP vem sendo largamente usado desde que surgiu por volta de 1980 e, de fato, se tornou o padrão em gerenciamento de redes de computadores (WU; ZHU; DENG, 2008). Além disso, diversas pesquisas propõem a aplicação do SNMP ao ambiente de redes de sistemas embarcados e IoT (CHOI; KIM; CHA, 2009; ETIENNE, 2014; KAKANAKOV; KOSTADINOVA, 2007; KURYLA; SCHÖNWÄLDER, 2011; LU *et al.*, 2015; STEFANOV, 2008).

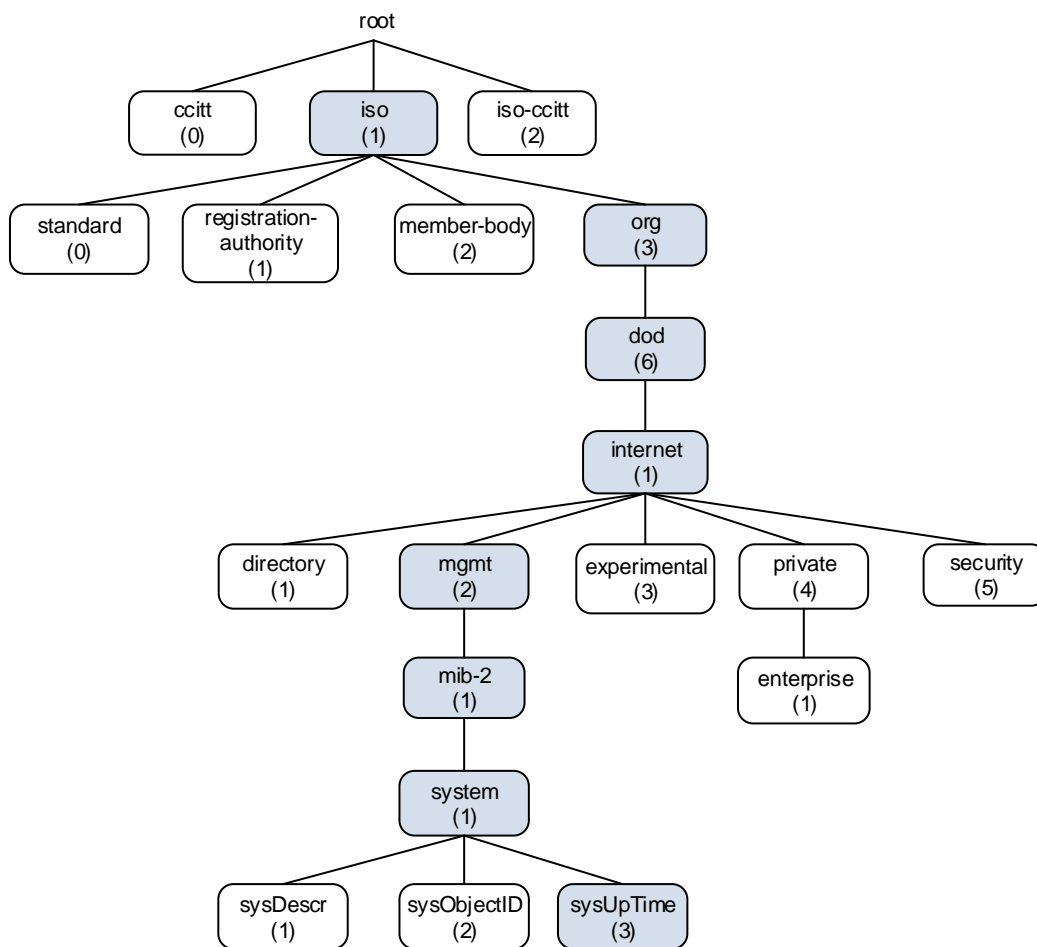
#### **2.10.1.1 MIBs**

Para possibilitar o acesso do servidor de gerenciamento às informações que se deseja monitorar e manter, é necessário que elas sejam identificadas e catalogadas. O SNMP resolve essa questão recorrendo à MIB. Neste trabalho, todas as operações em que o Servidor de Gerenciamento acessa o Dispositivo Gerenciado são feitas com base na identificação da informação desejada, e usa a MIB ou um modelo equivalente de localização

da informação.

A Base de Informação de Gerenciamento, em inglês chamada de *Management Information Base* (MIB), é uma base de registradores padronizada pelo protocolo SNMP que armazenam os dados de configuração de um cenário de tráfego e seu resultado. De forma simplificada, a MIB define a estrutura dos dados de gerenciamento de um dispositivo (ETIENNE, 2014).

Figura 2.10 – Parte da estrutura hierárquica da MIB



Fonte: próprio autor.

A MIB usa uma estrutura hierárquica de nomes chamada *namespace* contendo o identificador de um objeto, chamado de *Object ID* (OID), que pode ser lido ou escrito por operações SNMP (ETIENNE, 2014). Um identificador de objeto é montado usando segmentos de uma estrutura em formato de árvore definida pela ISO. Cada segmento da estrutura tem um nome, mas é representado por um número, e pode ser associado a uma organização (KAKANAKOV; KOSTADINOVA, 2007). Uma parte dessa estrutura é mostrada na Figura 2.10. O elemento raiz não tem nome e não é representado. Os identificadores do nível após a raiz pertencem às três principais organizações: *Comité*

*Consultatif International Téléphonique et Télégraphique* (CCITT), ISO, e a junção da ISO/CCITT. A MIB-II, padrão atual de MIB padronizada pela RFC-1213 (ROSE; MCCLOGHRIE, 1991), fica abaixo do nível “mgmt”, dentro da hierarquia da Internet (KAKANAKOV; KOSTADINOVA, 2007). Os identificadores usam os ramos mais apropriados da árvore. Assim, se quisermos referenciar o objeto “sysUpTime”, que tem o tempo total de funcionamento do dispositivo, usaremos o identificador OID “.1.3.6.1.2.1.1.3”, correspondente ao endereço “.iso.org.dod.internet.mgmt.mib-2.system.sysUpTime”. Os segmentos que formam o caminho desse OID na árvore MIB são destacados na Figura 2.10 em elementos com fundo escurecido.

Além disso, a árvore de identificadores de objetos é extensível. Os fabricantes podem definir seus próprios identificadores de objetos. Normalmente se usam os ramos “*experimental*” e “*private*” para criação de identificadores não padronizados pela ISO (KAKANAKOV; KOSTADINOVA, 2007).

## 2.10.2 Protocolo Zabbix

Os desenvolvedores do Zabbix também definiram um protocolo para comunicação com agentes de equipamentos que não suportam protocolos tradicionais como o SNMP. O protocolo Zabbix é extremamente simples e funciona sobre conexões TCP.

O agente Zabbix é implantado em um alvo de monitoramento e passa a fornecer dados relativos aos recursos locais. Os dados fornecidos podem ser enviados em formato limpo, quando se trata apenas de uma informação, ou podem ser encapsulados em objetos JSON, quando uma única mensagem contém várias informações de monitoramento. Toda comunicação do agente acontece sobre o protocolo TCP, através da porta 10050.

A identificação dos itens de configuração é chamada de chave. A lista completa de chaves suportadas pode ser obtida na documentação <sup>2</sup> oficial do Zabbix. Nessa documentação há uma lista básica das chaves<sup>3</sup>, uma lista classificada por plataforma<sup>4</sup> e uma lista de chaves específicas da plataforma Windows<sup>5</sup>.

---

<sup>2</sup> Disponível em <https://www.zabbix.com/documentation/3.0/start>. Acesso em abril de 2017.

<sup>3</sup> Disponível em [https://www.zabbix.com/documentation/3.0/pt/manual/config/items/itemtypes/zabbix\\_agent](https://www.zabbix.com/documentation/3.0/pt/manual/config/items/itemtypes/zabbix_agent). Acesso em abril de 2017.

<sup>4</sup> Disponível em [https://www.zabbix.com/documentation/3.0/pt/manual/appendix/items/supported\\_by\\_platform](https://www.zabbix.com/documentation/3.0/pt/manual/appendix/items/supported_by_platform). Acesso em abril de 2017.

<sup>5</sup> Em [https://www.zabbix.com/documentation/3.0/pt/manual/config/items/itemtypes/zabbix\\_agent/win\\_keys](https://www.zabbix.com/documentation/3.0/pt/manual/config/items/itemtypes/zabbix_agent/win_keys). Acesso em abril de 2017.

O protocolo pode funcionar de duas formas, ou em dois tipos de verificação: verificação passiva e verificação ativa.

- 1) **Verificação passiva:** é uma requisição simples de dados. O servidor pede um dado, carga da CPU, por exemplo, e o agente envia o resultado de volta ao servidor:
  - a. O servidor abre uma conexão TCP com o agente;
  - b. O servidor envia “cpu.load\n”;
  - c. O agente lê a requisição, obtém a informação e responde com “<HEADER><DATALEN>75”
  - d. O servidor processa o dado recebido, “75” nesse caso;
  - e. O servidor encerra a conexão TCP.
- 2) **Verificação ativa:** requer um processamento mais complexo. Nesse caso o agente deve inicialmente perguntar ao servidor quais dados ele deseja. O servidor envia a lista de informações desejadas. A partir disso, o agente passa a enviar periodicamente os novos valores desses dados para o servidor:
  - a. O agente abre uma conexão TCP com o servidor;
  - b. O agente solicita a lista de itens a verificar;
  - c. O servidor responde com a lista de itens;
  - d. O agente processa a resposta e guarda a lista de itens;
  - e. A conexão TCP é fechada;
  - f. O agente inicia coleta periódica dos dados;
  - g. A cada coleta:
    - i. O agente abre uma conexão TCP com o servidor;
    - ii. O agente envia a lista de valores;
    - iii. O servidor processa os dados e envia um status de volta;

iv. A conexão TCP é fechada.

Há agentes Zabbix disponíveis para vários sistemas operacionais, como Windows, servidor e *desktop*, GNU/Linux, IBM AIX, Solaris, HP-UX, AIX, Família BSD e OS X (HORST; PIRES; DÉO, 2015).

O agente implementado neste trabalho usa o modo passivo de verificação, por sua simplicidade, que se traduziria em um código de *firmware* menor. A especificação do modo passivo do protocolo Zabbix é mostrada na Figura 2.11.

Figura 2.11 – Detalhes do protocolo Zabbix em modo passivo

**Requisição do servidor:**

<item key>\n

**Resposta do agente:**

<HEADER><DATALEN><DATA>

**Onde:**

<HEADER> - "ZBXD\x01" (5 bytes)

<DATALEN> - comprimento do dado (8 bytes).

O valor 1 será formatado como 01/00/00/00/00/00/00/00  
(oito bytes em HEX, número de 64 bits)

Fonte: documentação *online* do Zabbix<sup>6</sup>.

### 2.10.3 Protocolo MQTT

O *Message Queue Telemetry Transport* (MQTT) é um protocolo de mensagens apresentado por Andy Stanford-Clark, da IBM, e Arlen Nipper, da Arcom em 1999 e foi padronizado em 2013 (AL-FUQAHA *et al.*, 2015). Seu objetivo principal é conectar dispositivos embarcados com aplicações e *middlewares*.

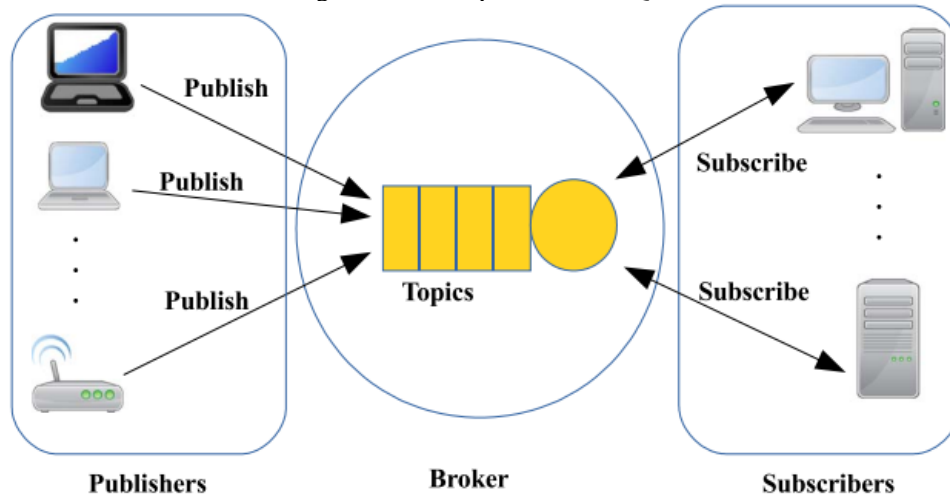
O protocolo MQTT não é anunciado como um protocolo de gerenciamento de redes, mas após uma análise detalhada fica claro que seu uso neste tipo de aplicação é perfeitamente possível. O termo telemetria já demonstra isso. A tecnologia de telemetria permite que coisas possam ser medidas ou monitoradas à distância (LAMPKIN *et al.*, 2012).

O MQTT trabalha sobre o protocolo TCP e utiliza o padrão *publish/subscribe*. Ele foi projetado para ser aberto e de fácil implementação, com a possibilidade de milhares de clientes serem atendidos por apenas um servidor (LAMPKIN *et al.*, 2012).

---

<sup>6</sup> Disponível em <https://www.zabbix.com/documentation/2.2/manual/appendix/items/activepassive>. Acesso em março de 2017.

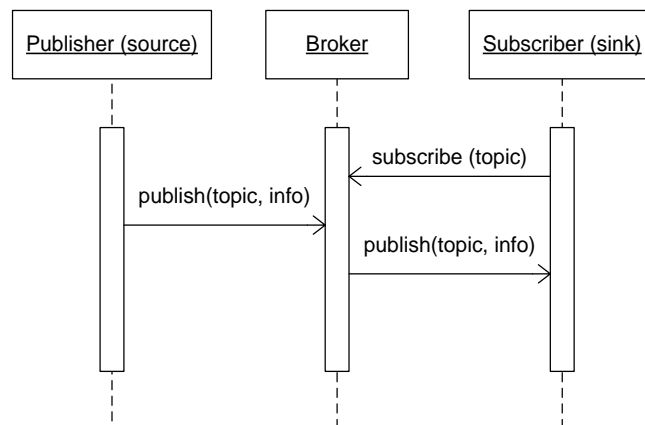
Figura 2.12 – Arquitetura do MQTT



Fonte: (AL-FUQAHA *et al.*, 2015).

O modelo do MQTT consiste de três componentes: *publisher*, *subscriber* e *broker*, conforme demonstra a Figura 2.12. Qualquer dispositivo interessado pode se registrar como um assinante (*subscriber*) de um tópico específico para ser informado pelo mediador (*broker*) quando algum editor (*publisher*) publicar no tópico de interesse. Após isso, sempre que o *publisher* transmitir uma informação, ela será enviada aos *subscribers* interessados através do *broker* (AL-FUQAHA *et al.*, 2015). A Figura 2.13 ilustra a sequência desse processo.

Figura 2.13 – Processo *publish/subscribe* usando pelo MQTT



Fonte: próprio autor.

Algumas características tornam o MQTT ideal para uso em dispositivos de recursos limitados. Ele é adequado para uso em situações onde a rede tenha baixa largura de banda ou onde haja alta latência e com dispositivos que possam ter capacidades limitadas de processamento e de memória (LAMPKIN *et al.*, 2012). O protocolo é especificado com base em alguns conceitos básicos que objetivam garantir a entrega das mensagens enquanto as

mantém o mais leve possível:

- **Modelo *publish/subscribe*:** o protocolo é baseado no princípio de publicar mensagens e assinar tópicos;
- **Tópicos e assinaturas:** as mensagens no MQTT são publicadas para tópicos, que podem ser entendidos como áreas de interesse. Os tópicos costumam ser formados por palavras separadas por uma barra (/) e que se assemelham a caminhos. Os clientes, por sua vez, podem assinar tópicos específicos ou podem usar coringas, como os sinais # e \*, para receber mensagens de uma variedade de tópicos relacionados;
- **Qualidade de serviço (QoS):** o MQTT define três níveis de qualidade de serviço – *Quality of Service* (QoS), onde cada nível indica o esforço do servidor de garantir a entrega da mensagem;
- **Mensagens retidas:** o servidor mantém as mensagens mesmo após o envio para os assinantes. Se uma nova assinatura é submetida para o mesmo tópico, qualquer mensagem retida é enviada ao novo cliente assinante;
- **Sessões limpas e conexões duráveis:** quando um cliente se conecta ao servidor, ele informa o parâmetro *clean session*. Se o parâmetro for *true*, todas as assinaturas são removidas quando ele desconectar do servidor. Se for *false*, a conexão é tratada como durável, e as assinaturas permanecem após a desconexão. Nesse caso, mensagens subsequentes que chegarem com alto QoS serão armazenadas e enviadas após a conexão ser reestabelecida;

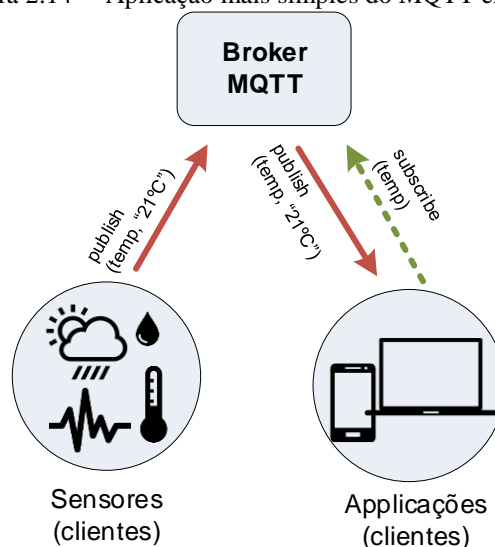
Lampkin (2012) faz uma interessante comparação entre os protocolos MQTT e HTTP (*Hypertext Transfer Protocol*), que é resumida na Tabela 2.1. A tabela usa diversos aspectos como base para a comparação. Em relação ao aspecto de projeto, Lampkin aponta que o MQTT é centrado no dado, enquanto o HTTP tem foco no documento que inclui dados e apresentação. No aspecto complexidade, Lampkin considera o MQTT mais simples. Lampkin também considera o tamanho da mensagem menor no MQTT comparado ao HTTP. Ele lembra que o MQTT tem três níveis de serviço enquanto o HTTP tem apenas um. E, na avaliação do aspecto de distribuição dos dados, aponta que o MQTT suporta três tipos de distribuição de dados, 1 para 0, 1 para 1 e 1 para n, enquanto o HTTP suporta apenas 1 para 1.

Tabela 2.1 – Comparação entre os protocolos MQTT e HTTP.

Aspecto	MQTT	HTTP
<b>Projeto</b>	Centrado no dado	Centrado em documento
<b>Padrão</b>	<i>Publish/subscribe</i>	<i>Request/response</i>
<b>Complexidade</b>	Simples	Mais complexo
<b>Tamanho da mensagem</b>	Pequena, com cabeçalho compacto	Grande
<b>Níveis de serviço</b>	3 níveis de QoS	Único nível de serviço
<b>Distribuição dos dados</b>	Suporta 1 para 0, 1 para 1, e 1 para n	Apenas 1 para 1

Fonte: (LAMPKIN *et al.*, 2012)

Figura 2.14 – Aplicação mais simples do MQTT em IoT



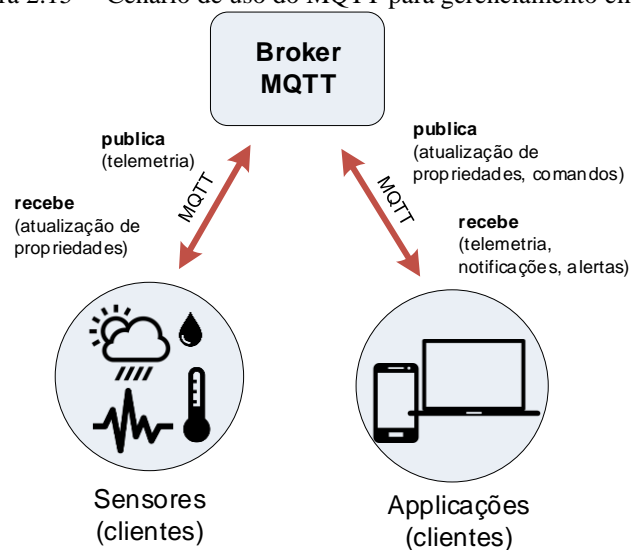
Fonte: próprio autor.

Uma aplicação simples do protocolo MQTT em IoT pode ser vista na Figura 2.14. Nesse cenário há sensores fornecendo informações que são aproveitadas e consumidas por aplicações. Para o MQTT, tanto os sensores quanto as aplicações são considerados clientes. Inicialmente as aplicações assinam o tópico desejado, nesse caso foi o “temp” que representa a temperatura. E sempre que os sensores publicarem no tópico “temp” a temperatura medida, o Broker enviará a mensagem a todas as aplicações que assinaram esse tópico.

Ao observar o modelo, a primeira impressão é que o protocolo MQTT é usado apenas para envio e coleta dos dados finalísticos do sistema. Entretanto, a IBM (TANG, 2015), empresa berço do protocolo, já sugere seu uso para um gerenciamento mais completo. Nesse cenário, os sensores poderiam assinar tópicos que seriam usados para as aplicações enviarem novos parâmetros e comandos aos sensores com o objetivo de alterar suas configurações e seu comportamento. A Figura 2.15 representa esse cenário, onde tanto os sensores quanto as aplicações assinam tópicos e publicam em tópicos.



Figura 2.15 – Cenário de uso do MQTT para gerenciamento em IoT

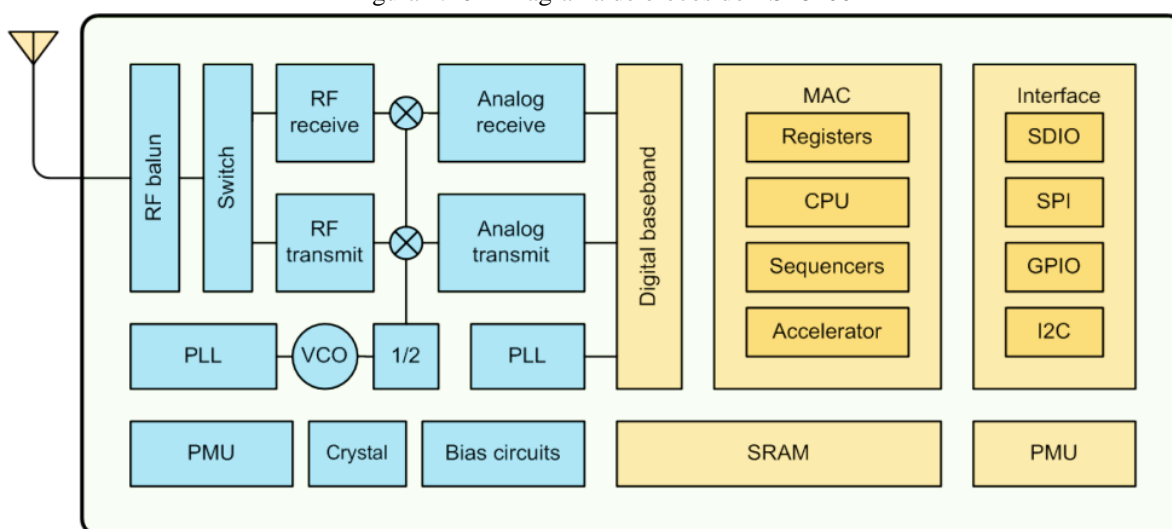


Fonte: próprio autor.

## 2.11 Microcontrolador ESP8266

O ESP8266 é um microcontrolador desenvolvido pela empresa chinesa Espressif Systems. O dispositivo é bem novo no mercado. Sua produção em escala e sua comercialização em volume iniciou em 2014. A chegada do ESP8266 no mercado tem movimentado os entusiastas da área, principalmente pelo seu baixo valor, na faixa de 4 dólares, e suas características, como microcontrolador de 32 bits, quase 100KB de memória RAM, 521KB de memória ROM, rede sem fio IEEE 802.11b/g/n integrada, suporte a modo de suspensão e tamanho bem reduzido. A Figura 2.16 mostra o diagrama de blocos com a arquitetura básica do ESP8266.

Figura 2.16 – Diagrama de blocos do ESP8266



Fonte: (ESPRESSIF SYSTEMS, 2017)

Há pouca documentação sobre o ESP8266. As principais fontes de informação são seus *datasheets*, disponibilizados no site da Espressif, os fóruns da comunidade de usuários e o livro de Kolban (2016). Apesar disso, nos últimos dois anos já surgiram pesquisas que tratam do dispositivo (FERNANDES *et al.*, 2017; OLIVEIRA, 2017; SANTOS *et al.*, 2016).

A Espressif o considera um *System On a Chip* (SoC) altamente integrado, de alto desempenho e uso eficiente de energia (ESPRESSIF SYSTEMS, 2017). Ele é considerado um SoC por ser um sistema completo em um único chip, incluindo microcontrolador, memória RAM, memória *cache* e toda parte de entrada e saída. Traz um processador Tensilica's L106 Diamond de 32 bits e diversas portas de entrada e saída, sendo uma delas com conversor analógico digital. Com capacidade completa e autocontida de rede Wi-fi, ele já possui conexão para antena interna e externa, amplificador de potência de sinal de transmissão com baixo ruído, filtros e módulos para gerenciamento de energia (ESPRESSIF SYSTEMS, 2017). A Tabela 2.2 traz mais detalhes sobre suas especificações.

Tabela 2.2 – Especificações do ESP8266.

Item	Valor
<b>Tensão de operação</b>	2,5V – 3,3V
<b>Corrente</b>	10µA – 170mA
<b>Corrente média de operação</b>	80mA
<b>Processador</b>	Tensilica L106 32-bit
<b>Memória RAM</b>	32K + 80K
<b>Memória ROM (Flash)</b>	512Kb (16MB máx.)
<b>Temperatura de operação</b>	-40°C – 125 °C
<b>Protocolos Wi-fi</b>	802.11 b/g/n/e/i
<b>Faixa de frequência</b>	2,4GHz ~ 2,4835GHz
<b>Antena</b>	PCB, Externa, IPEX
<b>Portas digitais (GPIO)</b>	17 (multiplexadas com outras funções)
<b>Portas digitais com PWM</b>	4
<b>Portas digitais com suporte IR</b>	2 (Tx e Rx)
<b>Portas A/D</b>	1
<b>Resolução A/D</b>	10 bits
<b>Dimensões</b>	5mm x 5mm

Fonte: (ESPRESSIF SYSTEMS, 2017)

Para o gerenciamento de energia a plataforma inclui recursos sofisticados para troca rápida entre os modos normal e de suspensão, além de ajuste de rádio adaptativo para operação em baixa potência. E, de acordo com seu datasheet, o sistema pode funcionar tanto como um microcontrolador escravo mais simples que provê dados para um *host*, como uma solução *standalone* que já possui a aplicação final (ESPRESSIF SYSTEMS, 2017).

Sua arquitetura de baixo consumo provê 3 modos de suspensão (*sleep mode*): suspensão do modem (*modem-sleep*), suspensão leve (*light-sleep*) e suspensão profunda (*deep-sleep*) (ESPRESSIF SYSTEMS, 2017). A Tabela 2.3 mostra as diferenças entre os modos de baixo consumo do ESP8266 e a corrente drenada em cada um desses modos.

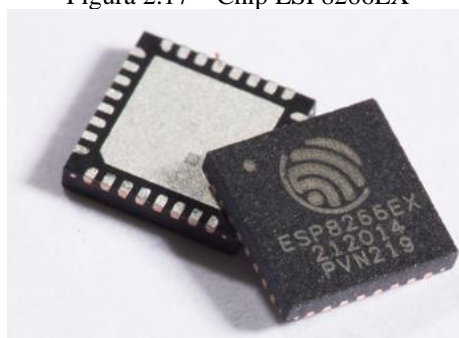
Tabela 2.3 – Diferenças entre os modos de suspensão do ESP8266.

Item	Modem-sleep	Light-sleep	Deep-sleep
Wi-fi	DESLIGADO	DESLIGADO	DESLIGADO
System clock	LIGADO	DESLIGADO	DESLIGADO
RTC	LIGADO	LIGADO	LIGADO
CPU	LIGADO	Aguardando	DESLIGADO
Corrente	15 mA	0,4 mA	~20 $\mu$ A

Fonte: (ESPRESSIF SYSTEMS, 2017)

O circuito integrado do ESP8266 vem em um pequeno encapsulamento de 5mm por 5mm e sua imagem pode ser vista na Figura 2.17. Entretanto, para sua programação e montagem final ainda é necessária outra placa de expansão (KOLBAN, 2016).

Figura 2.17 – Chip ESP8266EX



Fonte: Página do fornecedor<sup>7</sup>.

Alguns meses após seu lançamento, Espressif disponibilizou o primeiro Kit de Desenvolvimento de *Software* – *Software Development Kit* (SDK) para o ESP8266. Atualmente, já há ao menos três kits que permitem programar o *firmware* do ESP8266. Além do kit da Espressif, que oferece a linguagem C como opção de programação, também há o Kit NodeMCU, cujos programas podem ser escritos em linguagem LUA, e mais recentemente foi desenvolvida uma API para programar o ESP8266 usando o SDK do Arduino.

### 2.11.1 Estrutura da memória do ESP8266

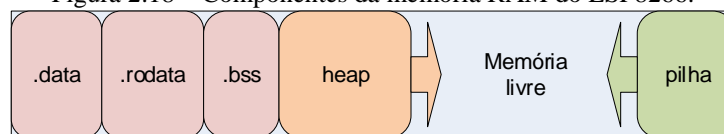
O ESP8266 possui ROM tipo Flash com 512Kbytes de espaço e 80Kbytes de memória RAM disponíveis. Essa memória RAM é preenchida por alguns componentes

<sup>7</sup> Disponível em: <http://www.electrodragon.com/product/esp8266ex-wifi-ic/>. Acesso em abr-2017.

estáticas e outros dinâmicos, como detalha a Figura 2.18. Abaixo segue a descrição de cada área:

- **.data**: usada para dados inicializados estaticamente. Os dados são copiados da memória ROM no momento em que a placa é iniciada;
- **.rodata**: usada para dados somente leitura, que não podem ser alterados durante a execução do *firmware*;
- **.bss**: usada para variáveis que não são iniciadas;
- **heap**: área alocada dinamicamente para armazenamento de variáveis criadas ao longo da execução do código;
- **pilha**: armazena informações sobre a sequência de execução das funções, além de variáveis locais criadas em cada função e que são descartadas ao sair da função.

Figura 2.18 – Componentes da memória RAM do ESP8266.



Fonte: próprio autor.

Como observado, a memória é ocupada no início por dados estáticos, dados somente leitura e variáveis que ainda não foram inicializadas. A área que sobra será ocupada pelo *heap*, que cresce de um lado, e pela pilha que cresce do outro lado (KOLBAN, 2016). A ocupação total da memória equivale a soma dos dados estáticos, dados somente leitura, *heap* e pilha de execução.

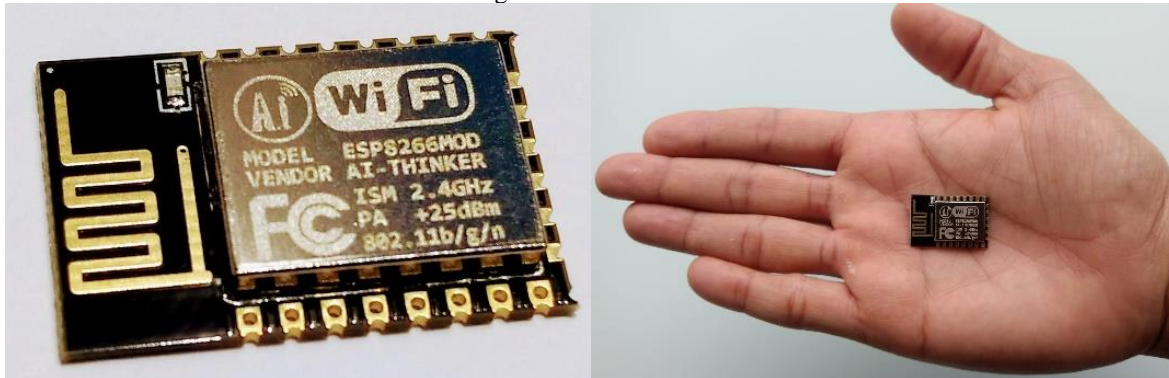
### 2.11.2 ESP-12

Diversos módulos fabricados por terceiros já incluem o chip em uma pequena placa de circuito impresso. O módulo ESP-WROOM-02, por exemplo, foi desenvolvido pela própria Espressif. Mas os módulos que mais se destacam no mercado são os módulos ESP-01 a ESP-14, da empresa AI-Thinker.

O módulo mais conhecido e mais usado atualmente é o ESP-12. O ESP-12 é um dos módulos mais flexíveis e consegue expor mais pinos de GPIO do ESP8266. A Figura 2.19

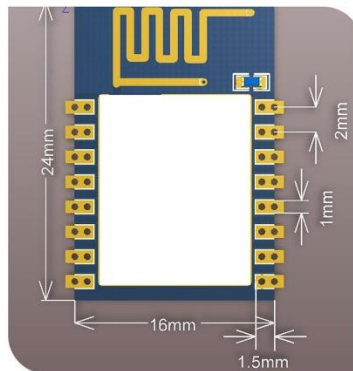
mostra o módulo ESP-12, que tem 16mm por 24mm de dimensões e que já traz uma antena de Wi-Fi impressa em sua placa, como pode ser visto na Figura 2.20.

Figura 2.19 – ESP-12



Fonte: próprio autor.

Figura 2.20 – Dimensões do ESP-12



Fonte: (AI-THINKER, 2015).

## 3. Materiais e método de pesquisa

---

Quanto aos procedimentos usados para a construção deste trabalho, trata-se de uma pesquisa experimental, devido à execução de experimentos controlados em laboratório com o objetivo de analisar, em um ambiente de rede real com dispositivos sensores e servidor de gerenciamento, o comportamento dos protocolos selecionados e implementados em agentes executados nestes dispositivos. Além disso, para apoiar o seu desenvolvimento, o trabalho também contempla procedimentos de uma pesquisa bibliográfica.

### 3.1 Método de pesquisa

A pesquisa experimental consiste em determinar um objeto de estudo, escolher as variáveis que podem influenciar este objeto e definir formas de controle e observação dos efeitos que a alteração dessas variáveis pode produzir no objeto (GERHARDT; SILVEIRA, 2009). As variáveis manipuladas são chamadas de variáveis independentes, e o seu efeito afeta as variáveis chamadas de dependentes.

Para alcançar os objetivos desta pesquisa experimental, foi necessária a realização das seguintes atividades:

- 1) Revisão da literatura;
- 2) Seleção da plataforma de *hardware*;
- 3) Escolha dos protocolos;
- 4) Planejamento do experimento;
- 5) Desenvolvimento do dispositivo sensor;
- 6) Desenvolvimento do *firmware* do dispositivo sensor;
- 7) Desenvolvimento dos agentes de gerenciamento;
- 8) Desenvolvimento do dispositivo medidor de consumo de energia;

- 9) Realização do experimento;
- 10) Análise e interpretação dos dados coletados;

### **3.1.1 Revisão da literatura**

Uma pesquisa bibliográfica é feita com o levantamento de referências teóricas analisadas e publicadas por meios escritos e eletrônicos (GERHARDT; SILVEIRA, 2009). Assim, a revisão da literatura que trata do tema tem como objetivo de elucidar os principais conceitos envolvidos no gerenciamento e no monitoramento de dispositivos que compõem as redes de sistemas embarcados e a Internet das Coisas.

### **3.1.2 Seleção da plataforma de *hardware***

Essa atividade engloba a pesquisa, análise e seleção de uma plataforma de *hardware* que permita a implementação e um ambiente de Internet das Coisas para realização dos experimentos. A plataforma selecionada deve atender aos critérios de um ambiente de IoT mas também deve suportar a maior quantidade possível de protocolos de gerenciamento de rede.

### **3.1.3 Escolha dos protocolos**

Seleção de protocolos de gerenciamento de rede, com base em padrões do mercado, que podem ser aplicados ao ambiente de Internet das Coisas e à plataforma de *hardware* escolhida.

### **3.1.4 Planejamento do experimento**

Essa atividade define como o experimento é conduzido. Aqui são definidos os passos do experimento, como são feitas as medições e quais materiais são necessários. Nessa atividade é definido o cenário do experimento, são definidos os demais itens da instrumentação e a seleção das variáveis a serem examinadas, os dados a serem coletados e quantas execuções serão realizadas.

### **3.1.5 Desenvolvimento do dispositivo sensor**

A atividade contempla o projeto, prototipação e construção do dispositivo sensor que

faz o papel de objeto inteligente. Aqui são definidas as funções que o dispositivo terá e as informações que ele poderá fornecer para o seu monitoramento. Do seu projeto resulta um diagrama esquemático do circuito e um diagrama de prototipação. A partir dos diagramas do projeto uma lista de componentes é gerada. Com base nessa lista os componentes podem ser adquiridos e o dispositivo pode ser montado.

### **3.1.6 Desenvolvimento do *firmware* do dispositivo sensor**

Nessa atividade deve ser desenvolvida a camada de *software* que será incluída no dispositivo sensor. Essa camada deve ter, basicamente, a função de coletar por meio de sensores algumas informações a respeito do ambiente, como **temperatura** e **umidade**. Isso diz respeito a atividade finalística do dispositivo. Porém, além disso, o *firmware* deve estar preparado para receber os agentes de gerenciamento.

### **3.1.7 Desenvolvimento dos agentes de gerenciamento**

Aqui são implementados os agentes de gerenciamento que suportam os protocolos selecionados na pesquisa. Será um agente para cada protocolo selecionado. O agente deve poder ser adicionado ao *firmware* para incluir a capacidade de gerenciamento remoto com a obtenção dos dados de monitoramento.

### **3.1.8 Desenvolvimento do dispositivo medidor de energia**

A atividade contempla o projeto, prototipação e construção do dispositivo que faz a medição de energia elétrica consumida por outro aparelho em funcionamento. O medidor é projetado levando em conta as necessidades esperadas do ambiente experimental, como faixa de tensão e de corrente, quantidade de amostras e forma de captura dos dados. Assim como acontece com o desenvolvimento do dispositivo sensor, o projeto do medidor resulta em um diagrama esquemático do circuito e um diagrama de prototipação. A partir dos diagramas do projeto uma lista de componentes é gerada, os componentes são adquiridos e o dispositivo pode ser montado.

### **3.1.9 Realização do experimento**

Nesta atividade o experimento foi executado e feita a coleta dos dados relativos às métricas definidas. Foram feitas as trocas dos protocolos e realizadas novas medições para



cada protocolo incluído no dispositivo. Todos os dados coletados foram anotados para posterior análise.

### **3.1.10 Análise e interpretação dos dados coletados**

Essa atividade contempla a análise, avaliação e interpretação dos dados que foram coletados na atividade anterior. Aqui foi feita uma análise crítica, identificando razões pelas quais os agentes consumiram memória do dispositivo.

## **3.2 Instrumentação**

Para a implementação do trabalho proposto, foi necessária a utilização dos materiais e recursos trazidos nesta seção.

### **3.2.1 Zabbix**

Conforme visto na seção 2.9, o Zabbix é um *software* de código aberto mas de nível empresarial, projetado para o monitoramento em tempo real de milhões de métricas coletadas a partir de dezenas de milhares de servidores, máquinas virtuais e dispositivos de rede.

Para uma escolha adequada do servidor de gerenciamento usado na pesquisa, seria recomendado fazer uma seleção criteriosa entre as opções disponíveis no mercado. Telesca (2014), da Organização Europeia para a Pesquisa Nuclear – *European Organization for Nuclear Research* (CERN), fez a avaliação de várias ferramentas de monitoramento para selecionar uma delas. A seleção inicialmente levou em conta ferramentas que tinham licença livre e que eram de código aberto. Em seguida, foram filtradas as ferramentas que tinham suporte ao protocolo SNMP, alertas, gatilhos, monitoramento distribuído, agrupamento lógico e grande comunidade de usuários. A seleção resultou em quatro ferramentas: Icinga, Cacti, Zenoss e Zabbix. A essa lista foram acrescentadas mais duas ferramentas. A ferramenta de monitoramento chamada MonALISA, que foi desenvolvida especificamente para a CERN, foi incluída. E a Splunk também foi considerada por estar sendo avaliada pela CERN-IT. Todas as ferramentas também possuem interface *Web* e controle de acesso.

Telesca (2014) definiu critérios de avaliação, e para estes foram dadas notas a cada ferramenta. As características avaliadas são detalhadas a seguir:

- **Coleta de dados:** maneira como os dados são coletados;
- **Representação gráfica:** refere-se às funcionalidades gráficas nativas da ferramenta;
- **Escalabilidade:** indica o número máximo de equipamentos que podem ser monitorados de forma eficiente;
- **Extensibilidade:** quão fácil é estender as funcionalidades da ferramenta;
- **SNMP:** capacidade de monitorar dispositivos usando o protocolo SNMP;
- **Documentação/Comunidade de usuários:** indica a facilidade de encontrar informações ou solução de problemas a partir de fontes da comunidade;
- **Granularidade máxima:** representa o menor intervalo de verificação;
- **Auto descoberta:** permite descoberta automática de dispositivos;
- **Grátis:** se a ferramenta é grátis ou tem custo de aquisição e uso.

Tabela 3.1 – Comparação entre ferramentas de monitoramento.

(a)

Name	Data gathering	Graphing (0-2)	Triggering (0-1)	Scalability (0-2) #hosts	Data Storage	Extensibility (0-2)
Icinga	Agent	0	1	1 - up to 1000	DB	2
Cacti	Server	2	0	1 - up to 1000	RRDtool - DB	2
Zenoss	Server	1	1	2 - 1000+	RRDtool - DB	1
Zabbix	Agent or Server	2	1	2 - 1000+	DB	2
Splunk	Agent	2	1	2 - 1000+	raw files	2
MonALISA	Agent	2	1	2 - 1000+	DB	2

(b)

Name	SNMP (0-2)	Documentation User Community (0-2)	Max Granularity (0-2)	Auto Discovery (0-2)	Free (0-1)	Total
Icinga	2	2	1-1 minute/metric	2	1	12
Cacti	2	2	1-1 minute/metric	1	1	12
Zenoss	1	1	1-1 minute/collector	2	1	11
Zabbix	2	2	2-No limit/metric	2	1	16
Splunk	2	2	2-No limit/metric	2	0	15
MonALISA	2	1	1-1 minute/metric	2	1	14

Fonte: (TELESCA *et al.*, 2014)

A Tabela 3.1 resume as pontuações das ferramentas em cada critério. A coluna total mostra a soma das notas dadas aos parâmetros. A avaliação resultou na seleção do Zabbix como ferramenta para a pesquisa de Telesca (2014). Levando em conta que todos os critérios definidos por Telesca também são adequados a esta pesquisa, também foi feita a escolha do Zabbix como ferramenta de gerenciamento e monitoramento deste trabalho.

### 3.2.2 Escolha do ESP8266

A escolha do ESP8266 ESP12 como dispositivo a ser usado no experimento se deu a partir da análise de suas características. Os recursos marcantes dos ESP8266 são:

- 1) Suporte a modo de suspensão com consumo na ordem de 20 $\mu$ A no modo de suspensão profunda;
- 2) Microcontrolador de 32 bits;
- 3) Memória RAM total de 112KB;
- 4) Memória ROM (Flash) de 512KB, expansível até 16MB


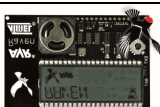



Também foi feita uma análise comparativa entre o ESP8266/ESP12 e dispositivos existentes no mercado como TMote Sky, AVR Raven, WisMote e Arduino BT, como pode ser visto na Tabela 3.2. Nessa comparação fica evidente a vantagem do ESP8266 nos quesitos “Processador”, “Memória RAM”, “Memória ROM” e “Padrões de rede sem fio”.

Por último, uma das maiores vantagens encontradas no ESP12 é o fato de ele já suportar conexões IEEE 802.11 b/g/n/e/i, que o capacita a trabalhar diretamente com a pilha TCP/IP, sem a necessidade de uso do 6LoWPAN. O suporte a conexões IEEE 802.11 b/g/n/e/i permite implementar toda a arquitetura de Internet das Coisas e seu gerenciamento dispensando o uso de outros elementos de rede. O dispositivo pode ser conectado diretamente à rede local e à Internet sem uso de nó de gateway para tradução de protocolos. Além disso, a camada de aplicação também fica independente da pilha 6LoWPAN, não sendo necessário usar protocolos de aplicação como CoAP e *Web Services*.

### 3.2.3 Placa NodeMCU

Como visto na seção 2.11, o ESP8266 é um circuito integrado bem pequeno que inviabiliza sua utilização diretamente pelo usuário final. Normalmente se adquire um dos modelos de suas placas prontas, e o modelo ESP-12 é uma das melhores opções do mercado pela quantidade de portas expostas e por custar pouco. Entretanto, fazer a prototipação e desenvolvimento da solução diretamente no ESP12 ainda não é trivial, pelas características dos seus contatos, que são muito próximos, e pelo seu tamanho reduzido. A utilização de kits de desenvolvimento para o ESP8266 facilita ainda mais a prototipação e construção de dispositivo baseado nesse chip.

Tabela 3.2 – Comparação entre dispositivos para IoT.

Item	Telos / TMote Sky / MTM-CM5000	AVR Raven	WiSMote	Arduino BT	ESP8266
<b>Fabricante</b>	AdvanticsSys	Atmel	Argo Systems	Vários	Espressif
<b>Tensão de operação</b>	2,1V a 3,6V	5V a 12V	3V	2,5V a 12V	2,5V a 3,3V
<b>Corrente</b>	5µA a 23mA	N/D	25,8 a 33,6mA	N/D	10µA a 170mA
<b>Corrente nominal</b>	21,8mA	N/D	30mA	N/D	80mA
<b>Modo suspensão</b>	Sim	Não	Sim	Não	Sim
<b>Corrente em suspensão</b>	10µA	N/D	2µA	N/D	20µA
<b>Processador</b>	Texas Instruments MSP430 F1611 16-bits RISC	Atmel ATmega3290P + ATmega1284P 8-bist RISC	Texas Instruments MSP430 5 series 16-bits	Atmel ATmega328 8-bits RISC	Tensilica L106 32-bits RISC
<b>Memória RAM</b>	10KB	16KB	16KB	2KB	32KB + 80KB
<b>Memória ROM</b>	48KB	4KB	128KB	32KB Flash 512 bytes EEPROM	512KB Flash (16MB máx.)
<b>Temperatura de operação</b>	-40°C a 85°C	-40°C a 85°C	-40°C a 85°C	-40°C a 85°C	-40°C a 125°C
<b>Padrões de rede sem fio</b>	802.15.4	802.15.4	802.15.4	Bluetooth	802.11 b/g/n/e/i
<b>Faixa de frequência</b>	2,4GHz ~ 2,485GHz	2,4GHz	2,4GHz	2,45GHz	2,4GHz ~ 2,4835GHz
<b>Antena</b>	PCB interna, externa, SMA	PCB interna	PCB interna, externa	PCB interna	PCB interna, externa, IPEX
<b>Portas digitais (GPIO)</b>	8	32 (multiplexadas com outras funções)	16	14	17 (multiplexadas com outras funções)
<b>Portas com PWM</b>	0	0	N/D	6	4
<b>Portas com suporte a IR</b>	0	0	0	0	2 (Tx e Rx)
<b>Portas A/D</b>	2	8	8	6	1
<b>Resolução A/D</b>	12 bits	10 bits	12 bits	10 bits	10 bits
<b>USB</b>	Sim	Módulo adicional	Não	Não	Não
<b>Sensor de temperatura</b>	Sim	Sim (NTC)	Sim	Não	Não
<b>Sensor de umidade</b>	Sim	Não	Sim	Não	Não
<b>Sensor de luz</b>	Sim	Não	Sim	Não	Não
<b>Sensor de áudio</b>	Não	Sim	Não	Não	Não
<b>Sensor de tensão de alimentação</b>	Sim	Sim	Não	Não	Não
<b>Acelerômetro</b>	Não	Não	Sim	Não	Não
<b>Mostrador</b>	Não	LCD		Não	Não
<b>Dimensões</b>	32 x 65 mm	78 x 54 mm	68 x 47 mm	81 x 53 mm	5 x 5 mm
<b>Custo</b>	90€ (~100U\$)	60U\$	130€ (~145U\$)	18U\$	5U\$
<b>Imagem</b>					

Fontes: datasheets dos produtos ESP8266X<sup>8</sup>, CM5000<sup>9</sup>, AVR Raven<sup>10</sup>, WiSMote<sup>11</sup>, Arduino BT<sup>12</sup>

<sup>8</sup> Disponível em: <https://espressif.com/en/support/download/documents>. Acesso em abr-2017.

<sup>9</sup> Disponível em: <https://www.advanticsys.com/shop/mtmcm5000msp-p-14.html>. Acesso em abr-2017.

<sup>10</sup> Disponível em: <http://www.atmel.com/tools/avrraven.aspx?tab=documents>. Acesso em abr-2017.

<sup>11</sup> Disponível em: <http://www.wismote.com/products.html>. Acesso em abr-2017.

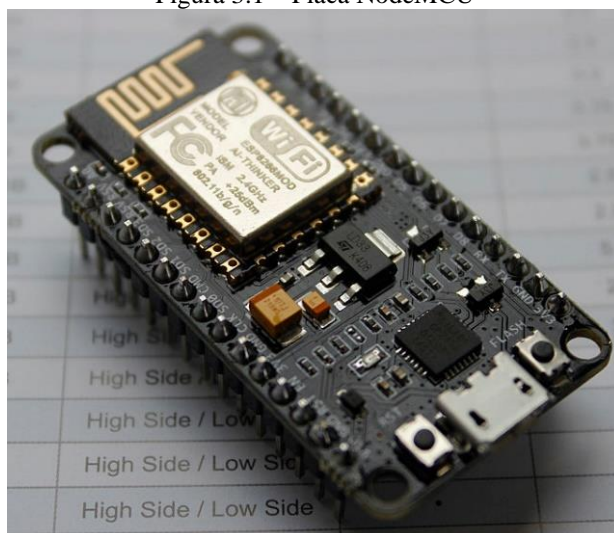
<sup>12</sup> Disponível em: <https://www.arduino.cc/en/Main/ArduinoBoardBT>. Acesso em abr-2017.

O NodeMCU é um kit de desenvolvimento aberto para ESP8266 bem completo. Ele inclui tanto um *firmware* baseado na linguagem de programação Lua, quanto uma placa que contém um ESP8266 ESP-12 e já traz um conector micro USB para ligação direta da placa com um PC, permitindo sua programação nos moldes da placa Arduino.

O ESP8266 trabalha com alimentação de 3,3V, mas o NodeMCU pode ser alimentado com tensões de até 9V. A placa NodeMCU possui um regulador de tensão que reduz a tensão da fonte para o nível requerido pelo ESP8266. Isso traz mais uma vantagem para a placa, que pode ser alimentada por porta USB, bateria ou conjunto pilhas.

A Figura 3.1 mostra que a placa NodeMCU já possui duas fileiras de pinos em distância padronizada, permitindo seu encaixe em placas de prototipação tipo *protoboard* ou mesmo em placas de circuito impresso prontas.

Figura 3.1 – Placa NodeMCU



Fonte: Página do projeto no GitHub<sup>13</sup>.

Pela facilidade de prototipação e desenvolvimento usando a placa NodeMCU, para a execução deste trabalho foi feita a opção por sua utilização. A escolha da placa não obriga o uso do ambiente de programação NodeMCU nem a linguagem Lua, sendo possível escolher qualquer um dos kits de desenvolvimento existentes. Nesse trabalho, foi utilizado o IDE Arduino como ambiente de programação.

### 3.2.4 Arduino IDE

O Arduino é uma plataforma aberta de computação física composta de um *hardware* com microcontrolador e um conjunto de *softwares* para programá-lo (MCROBERTS, 2011).

---

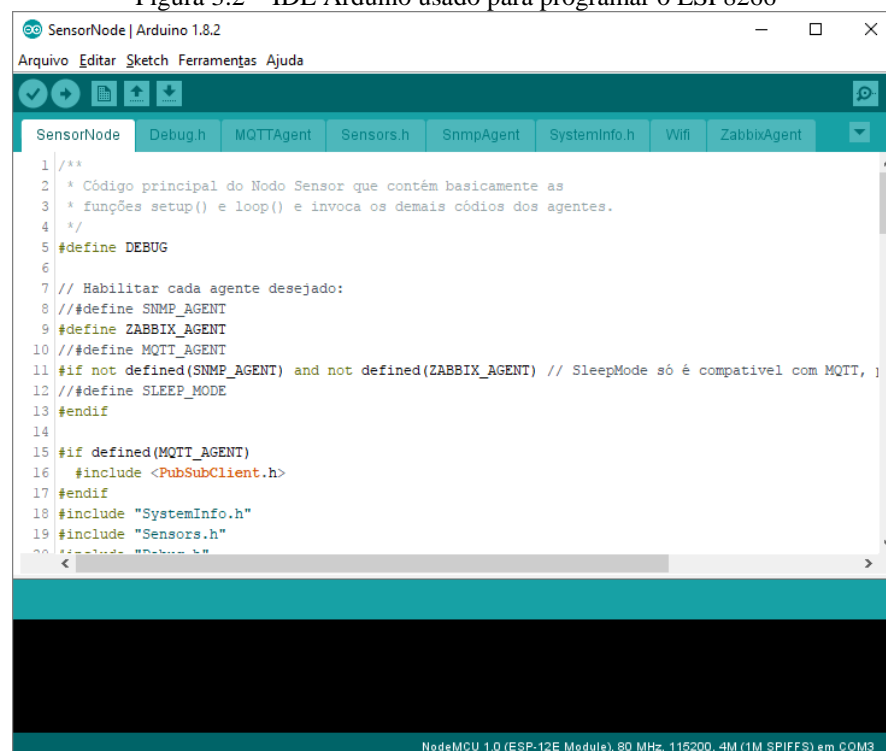
<sup>13</sup> Disponível em: <https://github.com/nodemcu/nodemcu-devkit-v1.0>. Acesso em abr-2017.

Para programar o Arduino é preciso usar o seu ambiente de desenvolvimento – *Integrated Development Environment* (IDE) no qual é escrito o código em uma linguagem baseada em C.

A plataforma do Arduino se tornou tão popular que foram criadas inúmeras comunidades de desenvolvimento em torno dela. Todo tipo de módulo pode ser adquirido para integração com o seu *hardware*. E uma quantidade imensa de bibliotecas foi desenvolvida para comunicação com todos os tipos de sensores e para diversos outros fins.

Como visto na seção 2.11, o primeiro Kit de desenvolvimento para o ESP8266 foi lançado pela Espressif e permite sua programação em linguagem C. Mas um grupo de desenvolvedores criou um projeto<sup>14</sup> que trouxe ao ESP8266 o suporte ao ambiente Arduino. Com isso é possível escrever programas para o ESP8266 usando todo o ecossistema do Arduino, aproveitando as diversas bibliotecas existentes.

Figura 3.2 – IDE Arduino usado para programar o ESP8266



Fonte: próprio autor.

Este trabalho fez uso do ambiente de desenvolvimento do Arduino para escrever o *firmware* e os agentes usados nos experimentos. A Figura 3.2 mostra uma imagem do IDE Arduino, que para usar com o ESP8266 precisa de algumas configurações adicionais.

<sup>14</sup> Disponível em <https://github.com/esp8266/Arduino>. Acesso em abril de 2017.

### 3.2.5 Mosquitto

O Mosquitto<sup>15</sup> é um *Message Broker* de código aberto que provê implementação cliente e servidor do protocolo MQTT (LIGHT, 2017). Ele implementa um *Broker* completamente funcional e isso permite que a pesquisa seja focada na análise do protocolo em si, deixando a complexidade do *middleware* a cargo do Mosquitto. Diversas pesquisas científicas relacionadas ao protocolo MQTT usaram o Mosquitto como *Broker* (AL-FUQAH et al., 2015; KODALI; SORATKAL, 2016; LAMPKIN et al., 2012; RAZZAQUE et al., 2016).

Além do ambiente acadêmico, o Mosquitto também é usado em outros projetos de código aberto, como o projeto de automação residencial OpenHab<sup>16</sup> e o projeto de rastreamento de localização pessoal OwnTracks<sup>17</sup>, além de produtos comerciais.

O projeto tem três partes:

- 1) mosquitto: servidor MQTT principal;
- 2) mosquitto\_pub e mosquitto\_sub: utilitários que facilitam a comunicação com o servidor;
- 3) Biblioteca cliente escrita em linguagem C.

Para a realização dos experimentos com o protocolo MQTT, foi utilizado apenas o servidor MQTT principal do Mosquitto. Seu uso se resume a iniciar o serviço executando um comando no sistema operacional. Em seguida, os clientes podem passar a se conectar no Mosquitto para assinar tópicos e publicar em tópicos.

### 3.2.6 Mqttwarn

Como mostrado na seção 2.10.3, o protocolo MQTT funciona na arquitetura *publish/subscribe*, na qual o cliente, no momento que desejar, publica em um Broker informações que podem ser aproveitadas por outros clientes. Entretanto, conforme visto na seção 2.9.1.1, o Zabbix funciona em um modelo em que o servidor consulta o agente instalado no dispositivo, solicitando as informações de monitoramento desejadas. Para compatibilizar as duas arquiteturas é preciso incluir um outro componente que faz a ponte

---

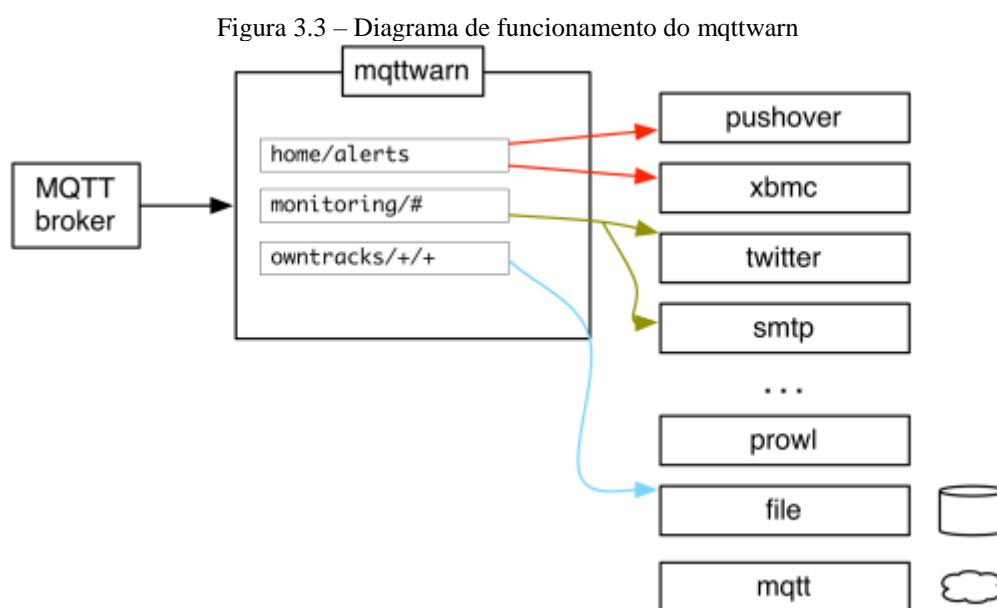
<sup>15</sup> Disponível em: <http://mosquitto.org/>. Acesso em mai-2017.

<sup>16</sup> <https://www.openhab.org/>. Acesso em mai-2017.

<sup>17</sup> <http://owntracks.org/>. Acesso em mai-2017.

entre o protocolo MQTT e o Zabbix. O mqttwarn é uma solução para esse componente.

O mqttwarn<sup>18</sup> é um *software* criado por Jan-Piet Mens, desenvolvido em Python, que pode atuar como um cliente MQTT e assinar qualquer tópico desejado em um Broker, repassando as mensagens publicadas nesses tópicos para *plugins* configurados no programa. Os *plugins* podem tratar as mensagens recebidas e enviar alertas para serviços diversos, como Twitter, e-mail, *media centers* e sistemas de arquivos. A Figura 3.3 representa o funcionamento do mqttwarn e seus *plugins*.



Fonte: Página do projeto mqttwarn<sup>19</sup> no GitHub.

Um dos *plugins* disponíveis foi desenvolvido para integrar o mqttwarn com o Zabbix. Ele permite tratar as mensagens recebidas e encaminhá-las para o Zabbix em forma de operação tipo TRAP. Os TRAPs recebidos no Zabbix podem ser usados para contabilizar indicadores e realizar o monitoramento de dispositivos. A arquitetura resultante desse arranjo é descrita na seção 4.2.

### 3.2.7 Arduino Nano

Para construir o medidor de consumo de energia elétrica usou-se o Arduino Nano<sup>20</sup>. Existem muitas versões de placas Arduino disponíveis, mas a Arduino Nano é uma das menores versões encontradas. A Figura 3.4 mostra o Arduino Nano e suas dimensões. A placa é baseada no microcontrolador ATmega328, da Atmel, e tem praticamente as mesmas

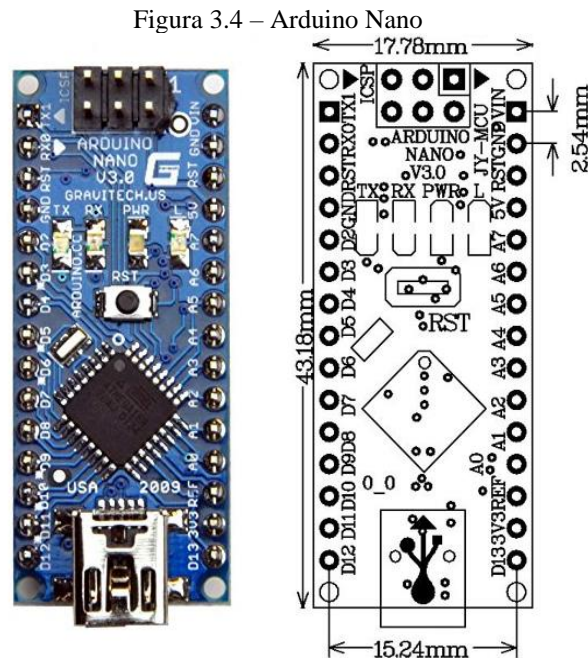
<sup>18</sup> Disponível em: <http://jpmens.net/2014/02/17/introducing-mqttwarn-a-pluggable-mqtt-notifier/>. Acesso em abr-2017.

<sup>19</sup> Disponível em: <https://github.com/jpmens/mqttwarn>. Acesso em abr-2017.

<sup>20</sup> Disponível em: <https://store.arduino.cc/usa/arduino-nano>. Acesso em abr-2017.



funcionalidades das placas mais tradicionais. Suas principais limitações em comparação com as placas Arduino tradicionais estão no número reduzido de portas de entrada e saída, e a falta de um conector de alimentação comum para fontes, sendo comumente alimentada diretamente pela porta USB



Fonte: Fórum oficial do Arduino<sup>21</sup>.

Alguns fatores foram considerados na escolha do Arduino Nano para construção do medidor de consumo de energia, elencados a seguir:

- 1) O baixo custo de aquisição da placa, podendo ser adquirida no Brasil por menos de 20 R\$;
- 2) A medição das grandezas desejadas necessita de apenas duas portas da placa;
- 3) A compatibilidade da placa com o módulo sensor INA219 escolhido, que usa o padrão de comunicação com barramento de dois fios I2C, através das portas SDA e SCL presentes na placa e no sensor;
- 4) Possibilidade de alimentação e transferência dos dados coletados pela porta USB.

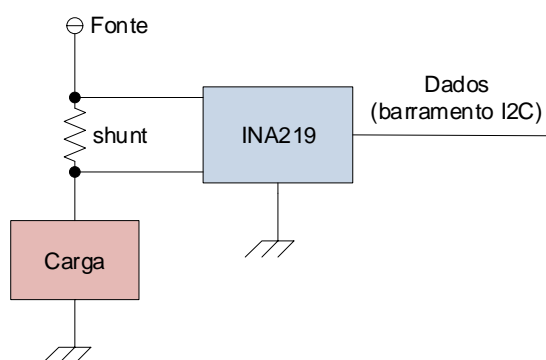
### 3.2.8 Sensor de Corrente INA219

O INA219, da Texas Instruments, é um sensor corrente elétrica por resistor *shunt* com

<sup>21</sup> Disponível em: <https://forum.arduino.cc/>. Acesso em abr-2017

precisão de 1% que usa uma interface compatível com o barramento I2C. A Figura 3.5 mostra o funcionamento do INA219. O dispositivo monitora a tensão do resistor *shunt*, a tensão de alimentação e a tensão da carga. Sabendo a resistência do resistor *shunt*, basta aplicar a lei de Ohm para encontrar a corrente elétrica que flui pela carga. As tensões e a corrente medidas são disponibilizadas por um barramento *Inter-Integrated Circuit* (I2C)<sup>22</sup>, criado pela Divisão de Semicondutores da Philips (atualmente NXP) no final da década de 70. A Tabela 3.3 lista os principais itens de suas especificações.

Figura 3.5 – Funcionamento do medidor de corrente INA219 baseado em resistor *shunt*



Fonte: autoria própria.

Tabela 3.3 – Especificações do sensor de corrente INA219.

Item	Valor
Tensão de alimentação	3 a 5,5 V
Faixa de tensão no barramento de teste	0 a 26 V
Corrente máxima de medição	3,2 A com resolução de 0,8 mA
Interface	I2C
Temperatura de operação	-40 °C a 125 °C

Fonte: *datasheet* do fabricante

É possível encontrar módulos que já trazem o chip INA219, um resistor *shunt* de 0,1 Ohm e mais alguns componentes passivos montados em uma placa, conforme mostrado na Figura 3.6. O módulo tem um amplificador de precisão que permite medir até três faixas de corrente com suas respectivas precisões. Com o ganho interno definido para o mínimo, é possível medir uma corrente máxima de 400 mA com resolução de 0,1 mA.

Diante da facilidade de obtenção do módulo, compatibilidade com a placa Arduino, precisão na leitura e velocidade de coleta, foi feita a escolha pelo INA219 como sensor de corrente a ser usado no medidor de consumo de energia elétrica.

<sup>22</sup> Disponível em: <http://www.nxp.com/docs/en/application-note/AN10216.pdf>. Acesso em abr-2017.

Figura 3.6 – Sensor de corrente INA219 da Texas Instruments



Fonte: Página do fornecedor<sup>23</sup>.

### 3.3 Implementação do Dispositivo Medidor de Energia

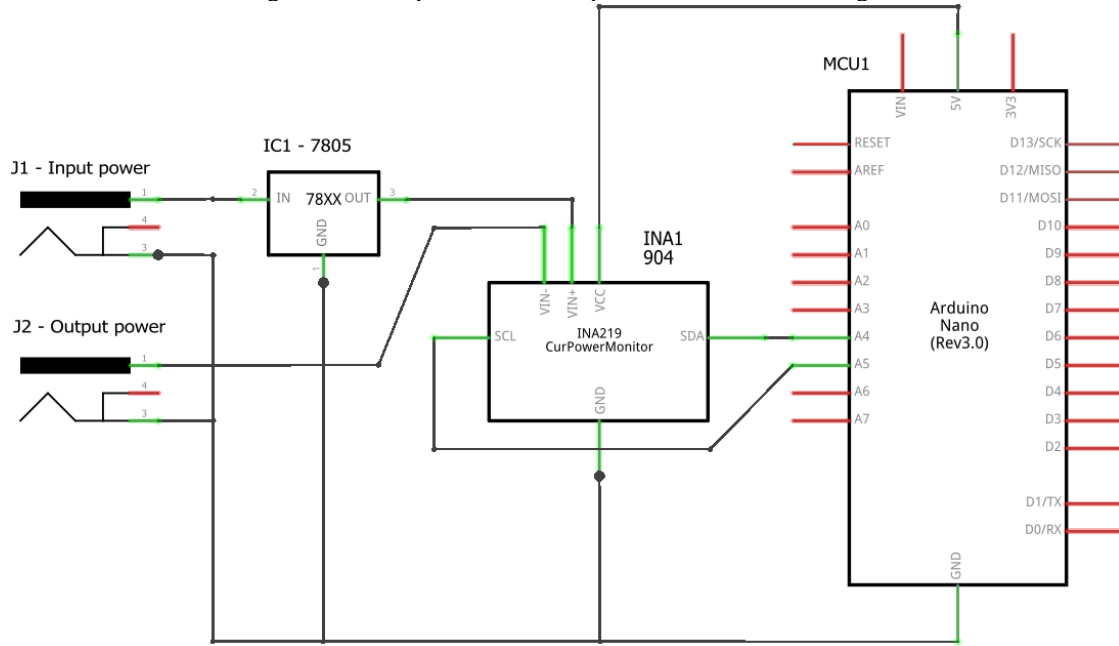
O dispositivo medidor de energia consumida foi implementado com base no sensor de corrente INA219 e uma placa Arduino Nano para coleta e processamento dos dados coletados. O circuito é alimentado por porta USB, que também é usada para transferência dos dados coletados para o PC, por meio de emulação de porta serial.

A Figura 3.7 mostra o esquemático do circuito medidor. A fonte de energia deve ser conectada em J1. A tensão de entrada é regulada por IC1, um regulador de tensão 7805 que fixa a tensão de saída em 5 V. O objetivo do regulador é tentar manter a tensão aplicada ao dispositivo sensor o mais estável possível, para que eventuais variações não afetem a potência final medida. A tensão regulada é então enviada ao sensor de corrente INA219 e segue para o conector J2 para alimentar o dispositivo sensor. Os dados são enviados pelos pinos SCL e SDA para o Arduino, que os captura, acrescenta algum tratamento, calcula a potência, carga e energia consumida, e envia as amostras formatadas para a porta serial. O PC deve monitorar essa porta para ter acesso às informações enviadas. Figura 3.8 mostra a prototipação do circuito e seu funcionamento pode ser visto na seção 3.4.2. O código fonte do firmware usado no Arduino pode ser obtido no GitHub<sup>24</sup>, conforme detalhado no APÊNDICE A.3

<sup>23</sup> Disponível em: <http://www.ebay.com/itm/191887834789>. Acesso em abr-2017.

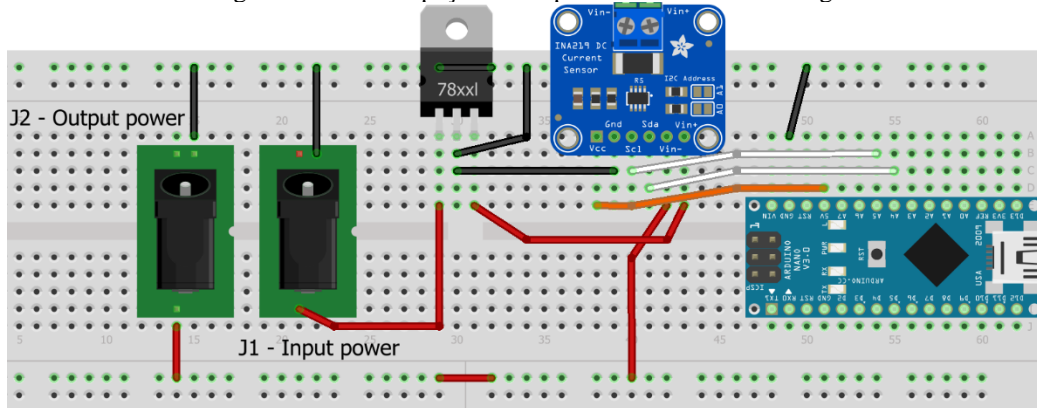
<sup>24</sup> Disponível em <https://github.com/levicm/EnergyMonitor/>.

Figura 3.7 – Esquemático do dispositivo medidor de energia



Fonte: próprio autor.

Figura 3.8 – Prototipação do dispositivo medidor de energia



Fonte: próprio autor.

## 3.4 Métricas usadas

Como o objetivo do trabalho é avaliar o impacto dos protocolos de gerenciamento nos recursos dos dispositivos gerenciados, as métricas usadas devem ser capazes de apontar o nível de consumo desses recursos.

### 3.4.1 Consumo de memória

Considerando que a memória é um dos recursos mais limitados dos dispositivos que são usados em IoT, a quantidade de memória RAM e de memória ROM ocupadas pelo agente são parâmetros importantes a serem avaliados (KURYLA; SCHÖNWÄLDER,

2011).

O *firmware* gravado no dispositivo inclui o código do programa que faz a função para a qual o dispositivo foi desenvolvido, mas também inclui o código do agente que permite o seu gerenciamento de forma remota. O código do *firmware*, incluindo o agente, é gravado na memória ROM tipo Flash, e este é carregado para a memória RAM quando é executado. A memória RAM guarda, além do código que será executado, os dados de constantes e variáveis usadas pelo programa. Assim, quanto menos espaço de código e de memória for ocupado pelo agente, mais espaço sobrar para o código principal do *firmware*.

Conforme detalhado na seção 2.11.1, a memória do ESP8266 é organizada em áreas que guardam dados estáticos e dados criados dinamicamente. De forma que, além do consumo de memória com variáveis estáticas, também é importante analisar o consumo de memória pela criação dinâmica de variáveis. Avaliar o consumo total de memória durante um período de execução permite observar a alocação dinâmica de variáveis e objetos feita pelo agente ao longo do tempo. Isso pode mostrar se o mesmo resiste a longas execuções. Também pode dar sinais sobre a eficiência de sua implementação, mostrando se o agente foi bem construído, não possuindo nenhum vazamento de memória.

Nesta dissertação de mestrado foi usado o método de Kuryla (2011) para medição da memória ocupada pelo código, tanto na memória ROM quanto na memória RAM. Para medir a ocupação do código na memória ROM, usou-se o relatório do compilador com o espaço ocupado pelo programa ao final na compilação. Para maior detalhamento das informações, foram feitas medições por camada de *software*. Para identificar o espaço ocupado por cada parte do código, foram sendo realizadas compilações com o programa vazio, para obter apenas o espaço ocupado pelo sistema operacional (kit de desenvolvimento), com o programa básico sem nenhum agente, para obter o espaço ocupado pelo *firmware* sem tratamento de gerenciamento, e com o programa incluindo cada agente individualmente, sendo anotados os tamanhos resultantes de cada compilação e feitas as diferenças entre estas. Ao final de cada compilação, o relatório do compilador já mostra o espaço ocupado pelo código do programa e o espaço que será ocupado pelos dados estáticos definidos no programa.

A medição da memória RAM ocupada fez uso de uma função específica do microcontrolador, disponível na plataforma por meio de chamada a `ESP.getFreeHeap()`, que retorna a quantidade de memória disponível em bytes. O processo de medição da memória

RAM ocupada por cada agente foi similar ao de medir a memória ROM. Foi carregado na placa o *firmware* sem nenhuma implementação e feita a medição da memória disponível, resultando no espaço ocupado apenas pela camada de *software* do kit de desenvolvimento. Em seguida, foi carregado apenas o programa básico, sem nenhum agente, e feita a medição de memória. Após, cada agente foi incluído individualmente no programa e carregado na placa, fazendo a medição de memória no início da execução, algumas no decorrer do funcionamento do agente e outra ao final do tempo estipulado para o experimento.

Usando esse processo foi possível medir as informações descritas a seguir:

- 1) Memória ROM ocupada pelo código do programa, identificando separadamente as camadas:
  - a. Sistema operacional: *bootloader* e toda API básica necessária ao kit de desenvolvimento;
  - b. Rede: porção do código responsável por toda a parte de conexão e tratamento de rede, incluindo os protocolos;
  - c. *Firmware* básico: código responsável pelas funções finalísticas do programa para as quais o dispositivo foi criado;
  - d. Agente: parte do código que faz a integração com o servidor de gerenciamento, expondo os itens de configuração desejados.
- 2) Memória RAM ocupada pelos dados estáticos, podendo também identificar separadamente as camadas descritas no item anterior;
- 3) Ocupação da memória RAM por dados estáticos e dinâmicos ao longo dos cinco minutos estipulados para a execução do experimento, conforme descrito na seção 5.1, que detalha o cenário do experimento.

### 3.4.2 Consumo de energia

O consumo de energia elétrica é uma das maiores preocupações quando se trata de dispositivos embarcados aplicados a IoT. Várias pesquisas se concentram nesse aspecto objetivando economizar a energia consumida por esses dispositivos restritos (ATZORI; IERA; MORABITO, 2010; DAGALE *et al.*, 2015; FENG; HUANG; SU, 2011).

A preocupação com o consumo de energia é particularmente importante quando a solução aplica dispositivos alimentados por baterias e que funcionam em campo, sem o uso da rede elétrica. Nesses casos, quanto menor o consumo de energia, maior a autonomia do dispositivo (MOUI; DESPRATS, 2011; WANG *et al.*, 2006).

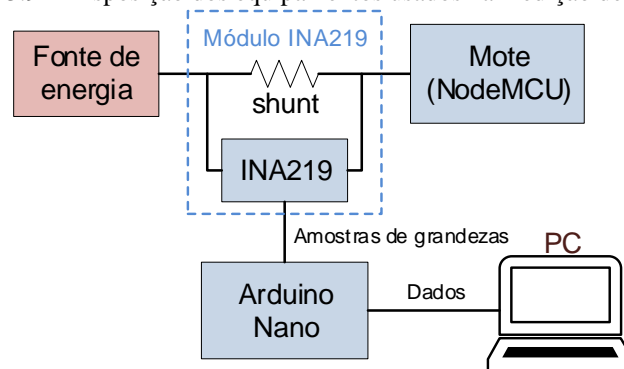
Convém trazer aqui, para deixar mais claro, algumas definições relacionadas ao consumo de energia. Os conceitos de potência e energia são detalhados a seguir.

**Potência:** Potência elétrica, ou dissipação de potência, é a medida instantânea de um índice de transferência de energia, ou o índice do trabalho realizado pela corrente elétrica. Ela é medida em Watts ou W. Seu cálculo é feito com o produto da diferença de potencial entre os terminais e a corrente que passa pelo dispositivo (KERRISON; EDER, 2015).

**Energia:** Energia, ou consumo de energia, é a medida do trabalho total realizado, ou o montante de potência que é dissipada com objetivo de alcançar o resultado desejado. Energia é a integral temporal da potência instantânea. Ela é tipicamente expressa em Joules ou J. Um sistema que sustenta uma potência de 1 Watt por 1 segundo terá consumido 1 Joule de energia. A energia também pode ser expressa em Watt-hora (Wh). 1 Watt-hora é a energia consumida por uma carga com potência de 1 Watt por 1 hora. 1 Wh é equivalente a 3600 J (KERRISON; EDER, 2015).

A medição da energia consumida em cada execução do experimento fez uso do dispositivo de medição construído com base no Arduino Nano e no sensor de corrente e tensão INA219.

Figura 3.9 – Disposição dos equipamentos usados na medição de energia

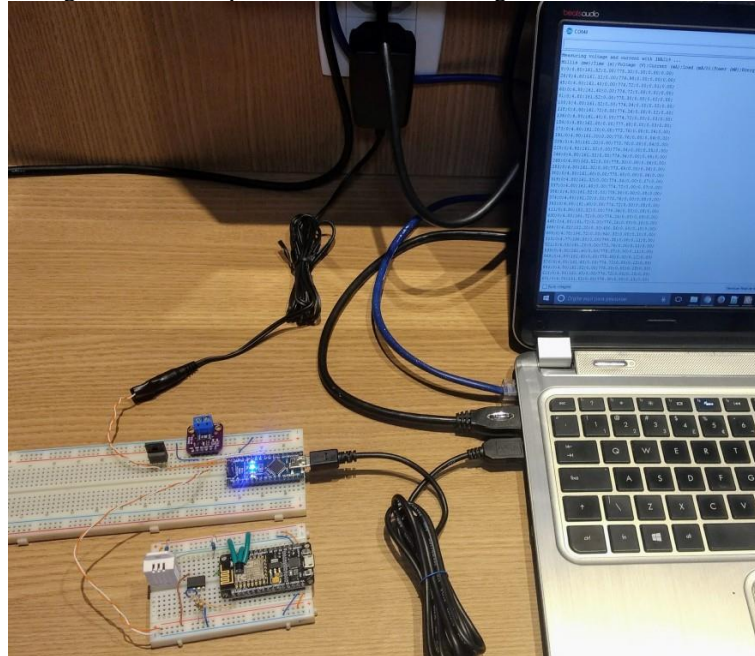


Fonte: próprio autor.

A Figura 3.9 ilustra a disposição dos equipamentos usados na medição de energia consumida. O medidor é colocado em série com o Mote e faz 53 coletas de indicadores elétricos por segundo. Os dados são enviados para um PC através da porta USB, que emula

porta serial. Uma console para porte serial é usada para visualização dos dados enviados, permitindo salvar o resultado em formato CSV, para registro e posterior processamento e análise.

Figura 3.10 – Dispositivo medidor de energia em funcionamento



Fonte: próprio autor.

Para cada execução do experimento, é feita uma limpeza na console de captura da porta serial, o dispositivo sensor e o dispositivo medidor são reiniciados simultaneamente, e os dados dessa execução passam a ser capturados. A Figura 3.10 mostra o circuito em funcionamento. Os seguintes dados são obtidos em cada coleta:

- 1) Momento da coleta, em milissegundos;
- 2) Tensão elétrica, em Volts (V);
- 3) Corrente elétrica, em miliamperes (mA);
- 4) Potência elétrica, em miliwatts (mW);
- 5) Energia elétrica consumida, em miliwatt-hora (mWh);
- 6) Carga elétrica transferida drenada, em miliampere-hora (mAh).



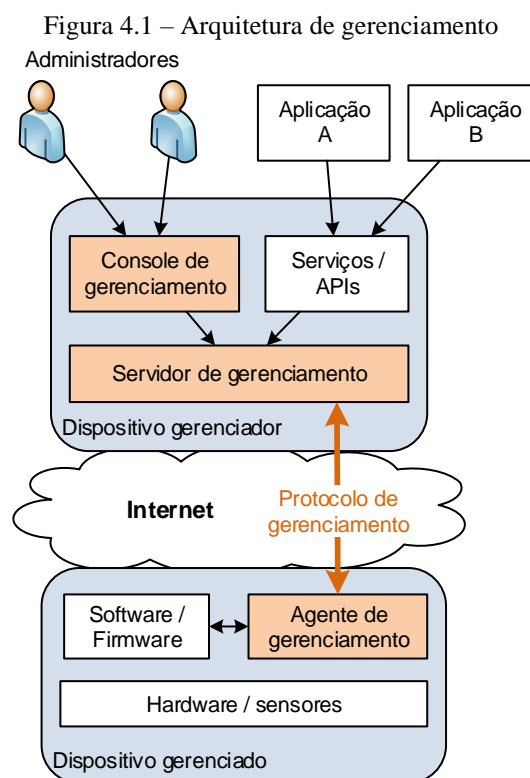
## 4. Desenvolvimento do ambiente experimental

---

Este capítulo detalha os passos realizados para o desenvolvimento do ambiente experimental. Aqui é mostrada a arquitetura usada no experimento, detalhando cada elemento desta arquitetura. Em seguida são mostrados os componentes e produtos definidos para cada elemento do ambiente experimental, bem como a disposição dos componentes nos equipamentos usados no experimento. O capítulo continua mostrando a instalação e configuração dos produtos usados. Por fim, é feito um detalhamento do desenvolvimento do dispositivo sensor, no que diz respeito a parte de *hardware* e de *software*.

### 4.1 Arquitetura de gerenciamento

Baseado na arquitetura de gerenciamento de redes em geral e nas arquiteturas das RSSFs e da IoT, tipicamente, seu gerenciamento é estruturado em uma arquitetura descrita na Figura 4.1.



Fonte: próprio autor.

A arquitetura prevê ao menos dois equipamentos e alguns elementos. Os dois equipamentos, mostrados na figura na cor azul, são o dispositivo gerenciador e o dispositivo gerenciado. Eles podem estar na mesma rede, em redes próximas, ou mesmo separados em redes distantes, conectados através da Internet, por exemplo. Podem existir mais de um dispositivo gerenciado, nos casos em que haja vários dispositivos sensores ou objetos inteligentes. A depender da topologia da rede, pode haver outros equipamentos de rede entre o dispositivo gerenciador e o gerenciado. Um exemplo disso é quando estamos gerenciando um nó de uma rede 6LoWPAN que, pela natureza da rede, normalmente precisa de um roteador de borda para permitir a ligação entre a rede de sensores e a Internet.

No dispositivo gerenciador são executados o servidor de gerenciamento e a console de gerenciamento. Nele também pode haver a exposição de serviços e API de gerenciamento para permitir que outras aplicações consigam integrar com o servidor de gerenciamento.

No dispositivo gerenciado fica toda camada de *hardware* e sensores que vão coletar dados do ambiente aonde são instalados, como também o *software/firmware* desenvolvido para a atividade fim do dispositivo, que são as coletas de dados. Nele fica o agente de gerenciamento, normalmente desenvolvido e carregado junto com o *software* do dispositivo, que se comunica com o Servidor por meio do protocolo de gerenciamento.

Os principais elementos dessa arquitetura, destacados na Figura 4.1 pelas cores azul e âmbar, são detalhados a seguir.

#### **4.1.1 Dispositivo Gerenciador**

O dispositivo gerenciador é um equipamento onde é executado o servidor de gerenciamento. Nele também pode ser executada uma console de gerenciamento, que proverá aos usuários uma interface amigável para permitir a execução das ações de gerenciamento. Nesse dispositivo podem ainda ser expostos serviços e Interfaces de Programação de Aplicação (APIs) que permitam a outras aplicações terceiras terem acesso às funcionalidades do servidor.

Neste trabalho, para realizar o papel de dispositivo gerenciador, foi feita a opção pela utilização de uma máquina virtual executada em um PC, por suas características de isolamento e independência.

### 4.1.2 Dispositivo Gerenciado

O dispositivo gerenciado é qualquer elemento de uma rede de sistemas embarcados que necessite de gerenciamento. Pode ser um Mote, um nó de uma WSN ou um dispositivo de um ambiente residencial inteligente. Além do próprio *software* existente nesse elemento que daria sua aplicação principal, também é necessário incluir um agente de gerenciamento, que será responsável por prover as principais funções de gerenciamento.

Como explicado nas seções 3.2.2 e 3.2.3, a opção aqui foi usar o ESP8266/ESP-12 como equipamento base para implementação do dispositivo gerenciado. O ESP8266 tem diversas características que o tornam uma excelente alternativa para implementação de sensores usados em Internet das Coisas.

### 4.1.3 Servidor e Console de Gerenciamento

O servidor de gerenciamento é o elemento responsável por requisitar, tratar e manter os dados relativos aos itens de gerenciamento dos dispositivos gerenciados. Ele provê um conjunto padronizado de funções, como monitoramento e controle, que permitem realizar o gerenciamento em si. Os dados mantidos pelo servidor de gerenciamento podem ser acessados por meio de uma interface com o usuário provida pela console de gerenciamento.

Após pesquisa, foi possível identificar que o Zabbix está sendo opção tanto em pesquisas científicas (TELESCA *et al.*, 2014), quanto no mercado, onde pequenas e grandes empresas têm optado por ele como solução de gerenciamento e como base para os seus Centros de Operação de Rede – *Network Operation Center* (NOC) (UYTTERHOEVEN, 2015).

O Zabbix pode ser usado tanto como servidor de gerenciamento quanto como console de gerenciamento, visto que já possui uma interface implementada em tecnologia *Web* que permite o acompanhamento dos indicadores por meio de valores descritivos ou por gráficos.

### 4.1.4 Protocolo de Gerenciamento

Um protocolo é um conjunto de regras sobre o modo como deverá acontecer a comunicação (BLANCHET, 2001). No gerenciamento, o protocolo determina como será feita a interface entre o servidor de gerenciamento e o agente de gerenciamento.

Como este trabalho trata do gerenciamento de dispositivos IoT, convém que os

protocolos escolhidos sejam direcionados a esse fim. Além disso, a escolha deve ser pautada por protocolos padronizados ou abertos, a fim de obter o máximo de reuso possível (MUKHTAR *et al.*, 2008).

Após análise de diversas pesquisas científicas sobre o tema e das tendências do mercado de IoT, foi feita a opção por comparar os protocolos SNMP, Zabbix e MQTT.

#### 4.1.5 Agente de Gerenciamento

Um agente de gerenciamento é um componente de *software* que reside no dispositivo gerenciado e atende a requisições do servidor de gerenciamento para prover informações sobre os itens de gerenciamento.

Neste trabalho foram desenvolvidos agentes com suporte aos protocolos avaliados e que podem expor as informações de monitoramento como tempo total de funcionamento, memória ocupada, identificação do chip, entre outras, detalhadas na seção 5.1.

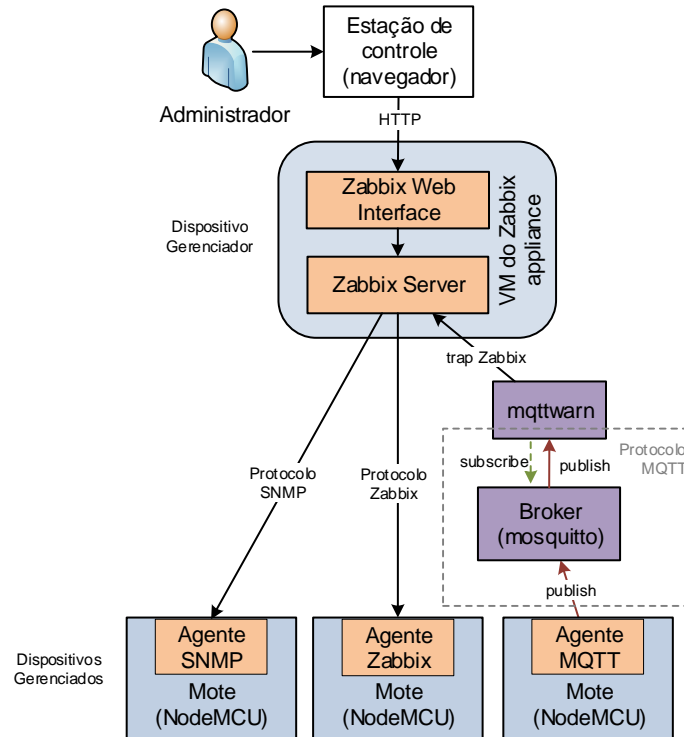
### 4.2 Ambiente experimental

Com base na arquitetura definida e na escolha de cada componente da arquitetura, o ambiente experimental pode ser estruturado e implementado. O desenho desse ambiente é representado pelo diagrama de componentes mostrado na Figura 4.2.

O Administrador usa uma estação qualquer da rede que possua um navegador *Web*. Usando o navegador é possível acessar a Interface *Web* do Zabbix que está sendo executada em uma máquina virtual tipo *Zabbix Appliance*. Nessa mesma máquina virtual também está sendo executado o servidor Zabbix, que acessa os Motes, feitos sobre o NodeMCU, através dos protocolos de gerenciamento. Isso já atende aos protocolos SNMP e Zabbix, mas não é suficiente para o funcionamento do protocolo MQTT. No MQTT, como já detalhado na seção 2.10.3, apenas o Broker funciona como servidor. Os demais elementos conectados ao Broker são clientes. Cada cliente estabelece uma única conexão com o Broker. Através dessa conexão, o cliente pode assinar tópicos ou ser avisado de um tópico que foi publicado. Esse comportamento exigiu a inclusão de mais dois elementos na arquitetura usada no trabalho. Um deles é o Mosquitto, funcionando como Broker. O segundo é o *mqttwarn*, que atua como um cliente que assina os tópicos das informações de monitoramento desejadas. Quando estas informações são publicadas pelo agente, o *mqttwarn* recebe as publicações e

as encaminha para o Zabbix em forma de mensagens TRAP.

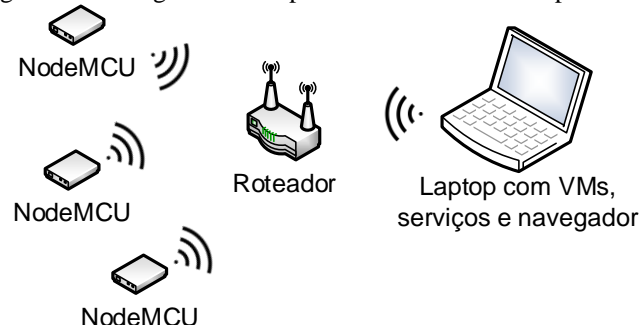
Figura 4.2 – Diagrama de componentes do ambiente experimental



Fonte: autoria própria.

A partir do diagrama de componentes do ambiente, foi possível identificar e estabelecer os equipamentos necessários ao ambiente experimental. O diagrama de dispositivos que mostra esses equipamentos é apresentado na Figura 4.3. Foram usados um laptop e três dispositivos sensores. Tanto a estação de controle quanto a máquina virtual do Zabbix *Appliance* podem ser executados no laptop. Já os dispositivos sensores foram construídos com base na placa NodeMCU, que é feita sobre o ESP8266.

Figura 4.3 – Diagrama de dispositivos do ambiente experimental

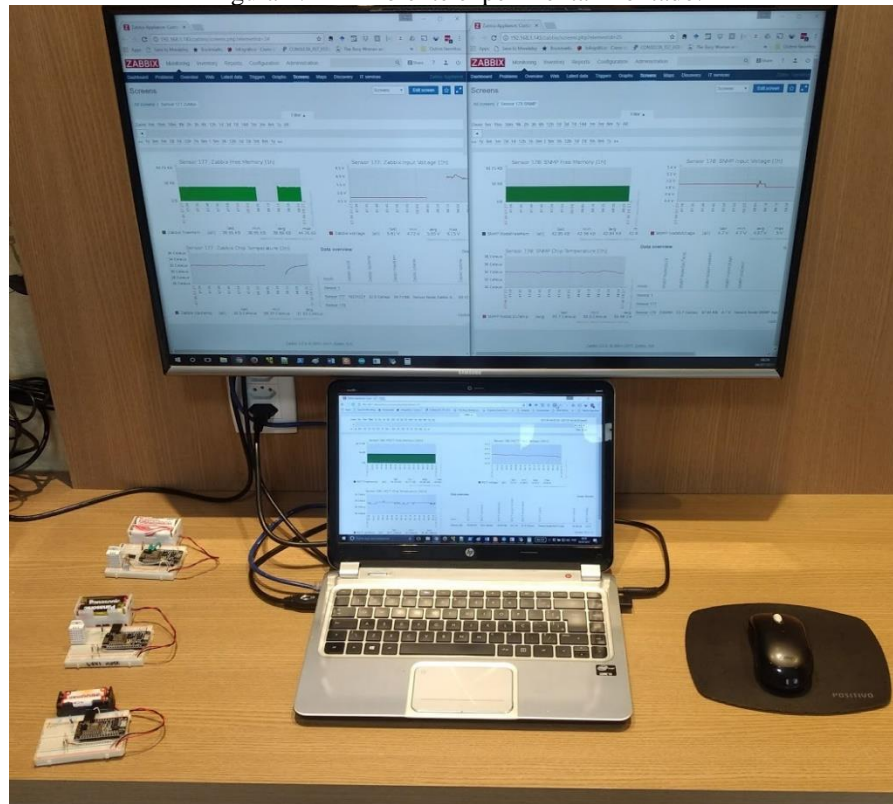


Fonte: autoria própria.

O resultado final da montagem do ambiente experimental pode ser visto na Figura 4.4. Foram construídos três Motes idênticos para funcionar como dispositivos gerenciados. Em cada Mote pode ser instalado um dos agentes implementados. Dessa forma, é possível

observar o funcionamento dos três protocolos, simultaneamente. Nesse ambiente foram executados os experimentos, feitas as coletas dos dados e analisados os resultados.

Figura 4.4 – Ambiente experimental montado.



Fonte: próprio autor.

A seguir são detalhadas as construções, implementações e configurações dos componentes necessários à montagem do ambiente experimental.

### 4.3 Instalação do Zabbix

Foi usado o Zabbix Appliance como máquina virtual para o Zabbix. O Zabbix Appliance é uma distribuição do produto feita em formato de máquina virtual, que pode ser obtida do *sítio Web* da ferramenta. Esse formato de distribuição tem a grande vantagem de já trazer todos os componentes do produto instalados e pré-configurados.

O Zabbix *Appliance* já traz instalado o Servidor Zabbix, a Interface *Web* e um banco de dados MySQL. A versão 3.2, usada no ambiente experimental, é baseada no Ubuntu Linux com a inclusão de alguns pacotes usados pelo Zabbix, e algumas ferramentas que ajudam os usuários durante as tarefas de configuração e manutenção. Além disso, o iptables é configurado para liberar algumas portas usadas pelo Zabbix, como a 10050 e 10051, e para bloquear qualquer situação diferente das previstas pelo serviço.

O processo para colocação do Zabbix Appliance em funcionamento se resumiu aos passos a seguir:

- 1) Baixar o *Zabbix Appliance* do sítio *Web* do Zabbix;
- 2) Importar a máquina virtual em uma ferramenta de virtualização;
- 3) Configurar o adaptador de rede da máquina virtual para funcionar em modo ponte, simulando sua conexão direta à rede da máquina hospedeira;
- 4) Executar a máquina virtual.

Como todos os serviços do Zabbix já estão configurados, tanto o servidor Zabbix quanto a Interface *Web* já funcionam logo após a execução da máquina virtual. Portanto, a interface já pode ser usada, acessando o endereço `http://<ip-zabbix-appliance>/zabbix`, a partir de qualquer navegador conectado à mesma rede.

## 4.4 Implementação do Dispositivo Gerenciado

O Mote é um bom exemplo de Dispositivo Gerenciado, por ser um típico dispositivo usado em ambientes de Internet das Coisas. Conforme visto na seção 2.3.1, um Mote é um dispositivo que já traz consigo alguns sensores e fonte de energia. Como o ESP8266 não traz esses elementos, foi necessária a construção de um dispositivo que tivesse alimentação própria e sensores para obter informações do ambiente, simulando assim um Objeto Inteligente. Além disso, foi acrescentado ao dispositivo a capacidade de prover medidas de desempenho e falha, permitindo um gerenciamento remoto útil e eficaz.

O dispositivo implementado tem as seguintes funções:

- 1) Prover informações do ambiente:
  - **Temperatura do ambiente**, em graus Celsius;
  - **Umidade relativa do ar**, em porcentagem;
- 2) Prover informações de falha e desempenho do dispositivo:
  - **Identificação do Chip**: permite diferenciar os vários dispositivos espalhados em um ambiente de rede;

- **Tempo total de funcionamento:** permite saber a quanto tempo o dispositivo está funcionando sem interrupção;
- **Memória RAM disponível:** permite acompanhar se o dispositivo está trabalhando com memória disponível suficiente para o seu bom funcionamento. Também permite avaliar tendências de ocupação de memória e permitir ações proativas em dispositivos que poderão ficar sem memória disponível dentro de algum tempo;
- **Temperatura do chip:** permite identificar se a temperatura do chip está dentro da faixa aceitável pelo fabricante;
- **Tensão de alimentação do dispositivo:** permite acompanhar a tensão da fonte de alimentação e identificar se a carga da bateria está próxima do final. Se combinada com a medição da corrente drenada pelo dispositivo, permitiria acompanhar o nível de carga da fonte de alimentação e a autonomia do dispositivo;

Figura 4.5 – Esquemático do dispositivo implementado

Fonte: autoria própria.



Tabela 4.1 – Lista de componentes do dispositivo implementado.

Identificação	Componente
DHT1	DHT22
DS1	DS18B20
R1	470k $\Omega$ Resistor
R2	470k $\Omega$ Resistor
R3	4.7k $\Omega$ Resistor
R4	4.7k $\Omega$ Resistor
U1	NodeMCU V1.0
U2	LM 358 Duplo Op-Amp
VCC1	Suporte de pilhas 4 x AAA

Fonte: autoria própria.

O sensoriamento do ambiente é feito com um DHT22, ou AM2302, da Aosong Electronics, que fornece tanto temperatura quanto umidade do ar digitalmente para o ESP8266. Ele utiliza um sensor capacitivo de umidade e um termistor para medir o ar circundante, ambos conectados a um controlador de 8 bits que produz um sinal digital serial no pino de dados (Data). O sensor permite medir temperaturas de -40° a 80° Celsius, e umidade na faixa de 0 a 100 %. Sua faixa de precisão para temperatura é de 0,1 graus, e para umidade é de 0,1%. Para a correta medição, é necessário usar um resistor de *pullup* de 4,7K $\Omega$  ligando o pino de saída de dados e a alimentação.

A temperatura do chip é medida com um DS18B20, da Dallas Semiconductors, que assim como o DHT22 também envia os dados em forma digital. Sua faixa de medição de temperatura é de -55°C a +125°C com precisão de 0,5° Celsius. O componente é posicionado junto ao chip do ESP8266 para refletir sua temperatura. Também é necessário ligar um resistor de *pullup* (R3) em sua saída de dados.

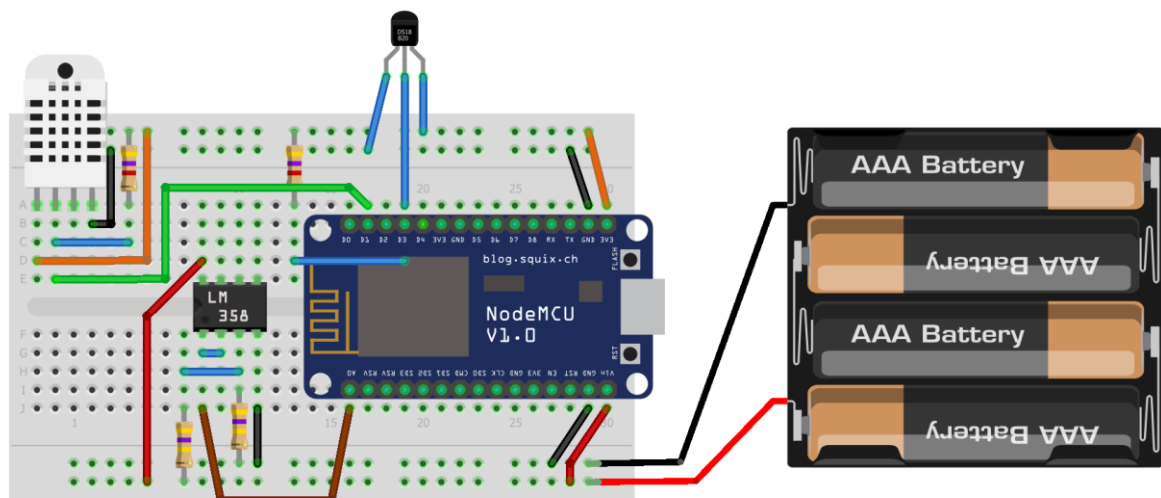
A última parte do circuito é o módulo de medição da tensão de alimentação. Como mostrado na seção 3.2.3, apesar do ESP8266 trabalhar com 3,3V, o NodeMCU pode ser alimentado com tensões de até 9V. O circuito deste trabalho foi projetado para ser alimentado com 4 pilhas, que somam até 6V. Como a porta analógica do ESP8266 só aceita tensões até 3,3V, a medição da fonte de alimentação é feita dividindo a tensão por dois e lendo esse valor na porta A0. O programa multiplica por dois o valor medido na porta e tem novamente o valor correto da tensão de alimentação. Isso permite medir tensões de alimentação entre 0V e 6,6V, cobrindo com sobra os 6V esperados na fonte.

O módulo de medição da tensão de alimentação é feito com um divisor de tensão resistivo e um amplificador operacional configurado como seguidor de tensão. Os resistores R1 e R2 dividem a tensão de alimentação por dois. Essa tensão dividida é enviada para o

amplificador operacional de ganho unitário, cuja saída é conectada diretamente à entrada A0 do ESP8266. O amplificador operacional funciona como *buffer* e evita que a entrada do ESP8266 altere uma das resistências do divisor de tensão e modifique a tensão lida. Os valores altos desses resistores evitam que essa parte do circuito gere um consumo indesejado da bateria. Os dois resistores de 470K $\Omega$  em série produzem uma resistência total de 940K $\Omega$ , que aplicada à fonte de alimentação de 6V gera uma corrente aproximada de 6,4 $\mu$ A.

Para diagramação do esquemático do dispositivo foi usada a ferramenta livre Fritzing<sup>25</sup>. A ferramenta, de código aberto, permite também a diagramação do protótipo do circuito em placas de prototipação, facilitando assim o processo de construção do dispositivo. O diagrama de prototipação é mostrado na Figura 4.6. Nele podem ser vistos todos os componentes dispostos na placa de prototipação, assim como os fios que devem ser usados para realizar as ligações adicionais entre os componentes.

Figura 4.6 – Prototipação do dispositivo implementado

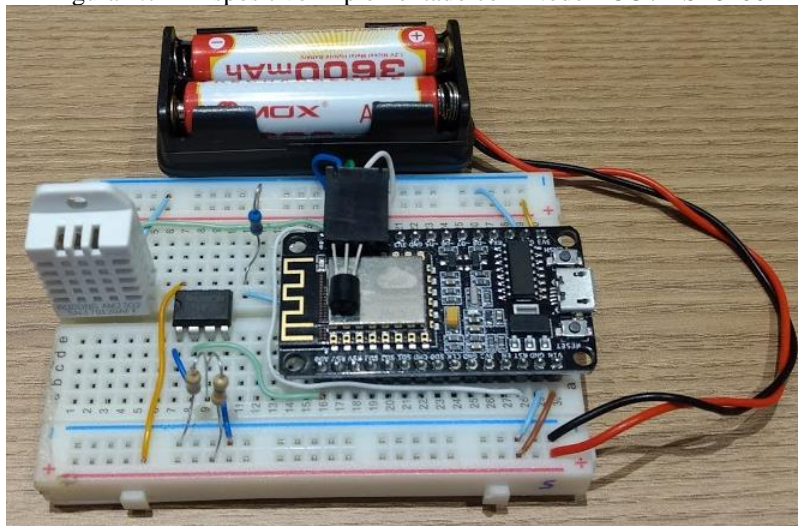


Fonte: autoria própria.

Após realizada a prototipação, o circuito foi montado sobre a placa, e o resultado é mostrado na Figura 4.7. Para verificar o correto funcionamento da montagem, alguns programas contendo códigos mais simples, de teste das partes do circuito, foram carregados na placa. Após a confirmação do funcionamento do dispositivo construído, esses códigos mais simples evoluíram e se tornaram as implementações do *firmware* e dos agentes. O processo de desenvolvimento do *software* do dispositivo é descrito nas seções que seguem.

<sup>25</sup> Disponível em <http://fritzing.org>. Acesso em maio de 2017.

Figura 4.7 – Dispositivo implementado com NodeMCU / ESP8266

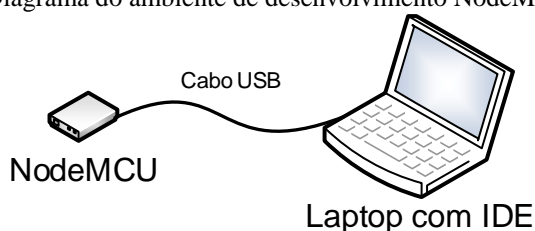


Fonte: autoria própria.

## 4.5 Programação da placa

O ambiente para desenvolvimento do NodeMCU é mostrado na Figura 4.8. A programação da placa, que envolve a implementação dos módulos que formam o programa completo é feita usando o ambiente de desenvolvimento do Arduino. O programa é escrito em linguagem C++ e, após compilado, é carregado na placa NodeMCU através de uma conexão USB com o laptop. O NodeMCU é reconhecido pelo computador como uma porta serial, que permite a gravação do programa compilado diretamente na memória Flash da placa.

Figura 4.8 – Diagrama do ambiente de desenvolvimento NodeMCU / ESP8266



Fonte: autoria própria.

Para que a o IDE do Arduino reconheça a placa NodeMCU são necessárias algumas configurações. É preciso acrescentar no Gerenciador de Placas do IDE um endereço de Internet<sup>26</sup> que provê todas as configurações específicas do ESP8266 e da placa NodeMCU. Isso feito, é possível selecionar a placa NodeMCU na lista de placas e realizar a compilação e envio do programa.

<sup>26</sup> Endereço das configurações do ESP8266 para o gerenciador de placas do IDE Arduino: [http://arduino.esp8266.com/stable/package\\_esp8266com\\_index.json](http://arduino.esp8266.com/stable/package_esp8266com_index.json)

Com o IDE configurado, o *firmware* foi implementado em vários códigos fonte. Os códigos fonte são separados em vários arquivos e bibliotecas para aumentar a modularização e com isso facilitar o entendimento e a manutenibilidade. Foram implementados os códigos fonte elencados a seguir:

1) **Firmware principal:**

- a. **SensorNode:** fonte principal do *firmware*. Onde a execução é iniciada;
- b. **Wifi:** rotina de conexão com a rede;
- c. **Debug:** macros para facilitar a depuração;
- d. **Sensors:** classe que encapsula a obtenção dos sensores de ambiente;
- e. **SystemInfo:** classe que encapsula a obtenção de informações sobre falha e desempenho do dispositivo;

2) **Uptime:** biblioteca que contabiliza o tempo de funcionamento do dispositivo;

3) **Agentes:**

a. **Agente SNMP**

- i. **SnmpAgent:** fonte do agente SNMP;
- ii. **AgentuinoWifi:** biblioteca que encapsula o protocolo SNMP;

b. **Agente Zabbix**

- i. **ZabbixAgent:** fonte do agente Zabbix;
- ii. **ZabbixPassiveWifi:** biblioteca que encapsula o protocolo Zabbix;

c. **Agente MQTT**

- i. **MQTTAgent:** fonte do agente MQTT.

O repositório com todos os códigos fonte desenvolvidos no experimento pode ser

acessado por meio da plataforma GitHub<sup>27</sup>, conforme descrito no “APÊNDICE A”.

## 4.6 Implementação do *firmware* principal

Para efeitos de organização, são considerados como códigos que fazem parte do *firmware* principal os fontes por onde a execução é iniciada e que tratam a rede, os fontes que fazem a leitura dos sensores e os fontes que obtém os dados de falha e desempenho.

O arquivo fonte principal é o “SensorNode”. Esse arquivo fonte inclui uma área de diretivas, algumas bibliotecas e três funções básicas. A área de diretivas contém alguns componentes `#define` que permitem informar ao compilador quais agentes serão incluídos na compilação. As bibliotecas são modularizações da rotina de obtenção das informações do sistema e da rotina de obtenção dos sensores de ambiente. Das três funções implementadas, uma, chamada de `printMemory()`, imprime na conexão serial as informações de memória que serão coletadas pelo experimento em intervalos de dez segundos para avaliar a evoluções da memória RAM ao longo de uma execução. As outras duas funções são a `setup()` e a `loop()`. A função `setup()` é executada uma única vez sempre que ESP8266 é iniciado e faz as configurações de início do programa, invocando as funções `setup()` equivalentes para cada agente que estiver ativo. E a função `loop()` é invocada continuamente pelo SDK para realizar as tarefas para as quais o *firmware* é destinada. Ela apenas invoca a função `printMemory()` para imprimir as coletas de memória, e em seguida invoca as funções `loop()` de cada agente habilitado.

O arquivo fonte “Wifi” contém todo o tratamento para conexão com a rede, sendo implementado com base na biblioteca “ESP8266WiFiMulti”, que permite configurar conexão com várias redes Wi-fi, de maneira que a placa possa ir realizando várias tentativas até que seja estabelecida uma conexão.

O arquivo fonte “SystemInfo” contém uma classe que trata a obtenção das informações do sistema como identificação do chip, tempo de funcionamento da placa, quantidade de memória livre, temperatura do chip e tensão de alimentação. Esse código usa as bibliotecas `OneWire` e `DallasTemperature`, para acesso ao componente sensor de temperatura digital DS18B20. E também usa a biblioteca “Uptime”, implementada neste trabalho para realizar a contabilização do tempo total de funcionamento do dispositivo.

---

<sup>27</sup> Disponível em <https://github.com/levicm/nes-management/>

O arquivo “Sensors” possui o código de uma classe de mesmo nome que encapsula tratamento para obtenção dos sensores ambientais que dão medidas de temperatura e umidade. Para isso ele usa a biblioteca DHT que facilita o uso do componente DHT22.

#### 4.6.1 Biblioteca Uptime

Para contabilizar o tempo total de funcionamento do dispositivo foi necessário o desenvolvimento de uma biblioteca que encapsulasse essa funcionalidade. A biblioteca foi chamada de “Uptime”, está disponível publicamente no GitHub<sup>28</sup>, conforme descrito no “APÊNDICE A”, e pode ser usada por qualquer pessoa que necessite da mesma funcionalidade.

A biblioteca “Uptime” contém apenas uma classe de mesmo nome, cuja implementação é essencialmente baseada na função `millis()`, da plataforma Arduino, que retorna o tempo decorrido em milissegundos. Os principais métodos da classe “Uptime” são descritos a seguir:

- **begin()**: inicia a classe;
- **compute()**: computa o tempo de funcionamento do dispositivo e guarda em variáveis;
- **totalSeconds()**: retorna o tempo total de funcionamento em segundos;
- **days()**: retorna a quantidade de dias de funcionamento do dispositivo;
- **hours()**: retorna as horas, dentro do dia, de funcionamento do dispositivo;
- **minutes()**: retorna os minutos, dentro da hora, de funcionamento do dispositivo;
- **seconds()**: retorna os segundos, dentro do minuto por `minutes()`, de funcionamento do dispositivo;
- **milliseconds()**: retorna os milissegundos, dentro do segundo retornado por `seconds()`, de funcionamento do dispositivo;

---

<sup>28</sup> Disponível em <https://github.com/levicm/nes-management/>

Sempre que é necessário verificar o tempo de funcionamento, o método `compute()` deve ser chamado para em seguida chamar os métodos que retornam o tempo total de funcionamento do dispositivo. Se, por exemplo, o dispositivo já está funcionando há 2 dias, 7 horas, 25 minutos, 17 segundos e 200 milissegundos, as chamadas aos métodos `days()`, `hours()`, `minutes()`, `seconds()` e `milliseconds()` retornarão respectivamente os valores 2, 7, 25, 17 e 200. Já o método `totalSeconds()` retorna o tempo total acumulado em segundos, que nesse exemplo é  $((((2 \times 24) + 7) \times 60) + 25) \times 60 + 17 = 199.517$  segundos.

## 4.7 Implementação dos agentes e seus protocolos

Os agentes de gerenciamento foram implementados na sequência: SNMP, Zabbix e MQTT. Para cada agente, foi necessário seguir os passos:

- 1) Procurar uma biblioteca pronta que já suporte o protocolo desejado;
- 2) Se encontrar biblioteca, verificar se já atende ou se precisa de modificação;
- 3) Se não encontrar biblioteca, implementar uma nova;
- 4) Implementar o agente fazendo uso da biblioteca encontrada, modificada ou implementada;
- 5) Configuração do Zabbix para capturar dados coletados do agente;
- 6) Realizar monitoramento do dispositivo através da console Zabbix;
- 7) Executar coleta dos dados.

Nas seções seguintes são detalhadas as particularidades da implementação de cada agente.

### 4.7.1 Agente e protocolo SNMP

Seguindo a sequência de passos descrita na seção anterior, inicialmente foi feita uma busca por uma biblioteca SNMP, que resultou na localização da *Agentuino*<sup>29</sup>. Entretanto, a biblioteca não suporta conexões de rede sem fio, requisito necessário quando se usa o ESP8266. Para resolver essa limitação, a *Agentuino* foi portada para uma nova biblioteca

---

<sup>29</sup> Disponível em <https://code.google.com/archive/p/agentuino/>. Acesso em março de 2017.

com o suporte à rede sem fio. Para isso, foi criada uma cópia da Agentuino e feitas alterações para incluir o suporte desejado. A nova biblioteca foi nomeada de AgentuinoWifi e disponibilizada no GitHub para uso público.

O agente SNMP foi implementado usando a nova biblioteca AgentuinoWifi. Sua implementação tem duas partes principais. A função `setupSnmpAgent()`, que faz a inicialização do objeto global da biblioteca, indicando a função do agente a ser chamada em caso de chegada de requisições SNMP. E a função `pduReceived()`, onde é feito o tratamento para identificar a operação SNMP e o OID solicitado pelo servidor. Se a operação for GET e o OID for suportado, o agente solicita à classe `SystemInfo` a informação requisitada, que em seguida é devolvida ao servidor.

Foi necessário fazer a escolha e a definição da MIB suportada pelo agente. Como detalhado na seção 2.10.1.1, o SNMP funciona a partir de uma árvore MIB, que forma o conjunto de identificadores de itens gerenciáveis do dispositivo, chamados de OID. Alguns OIDs que estão na especificação da MIB-II já estão alinhados com as informações de falha e desempenho que o dispositivo deste trabalho deve prover, já detalhadas na seção 4.4. Foram usados os OIDs “1.3.6.1.2.1.1.1.0” e “1.3.6.1.2.1.1.3.0” da MIB-II. E foram acrescentados mais quatro OIDs específicos do experimento. A lista de OIDs usados no agente é mostrada na Tabela 4.2.

Tabela 4.2 – Lista de OIDs implementados no agente SNMP.

OID	Caminho	Descrição
1.3.6.1.2.1.1.1.0	.iso.org.dod.internet.mgmt.mib-2.system.sysDescr	Retorna o nome do agente.
1.3.6.1.2.1.1.3.0	.iso.org.dod.internet.mgmt.mib-2.system.sysUpTime	Retorna o tempo total de funcionamento do dispositivo (em segundos).
1.3.6.1.2.1.1.9.1.0	.iso.org.dod.internet.mgmt.mib-2.system.node.nodeTypeCpuId	Retorna a identificação do microcontrolador.
1.3.6.1.2.1.1.9.2.0	.iso.org.dod.internet.mgmt.mib-2.system.node.nodeTypeFreeMem	Retorna a memória disponível no dispositivo (em bytes)
1.3.6.1.2.1.1.9.3.0	.iso.org.dod.internet.mgmt.mib-2.system.node.nodeTypeCpuTemp	Retorna a temperatura do microcontrolador (em graus Celsius).
1.3.6.1.2.1.1.9.4.0	.iso.org.dod.internet.mgmt.mib-2.system.node.nodeTypeVoltage	Retorna a tensão de alimentação do dispositivo (em Volts).

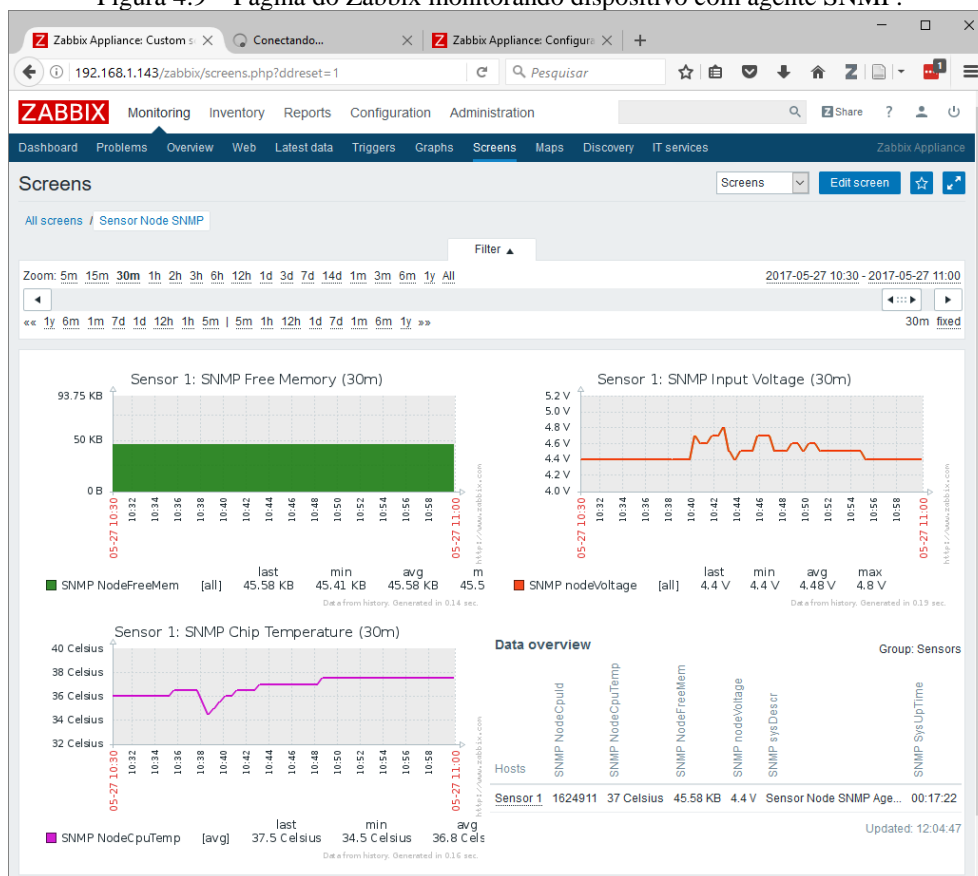
Fonte: autoria própria.

Com o agente implementado, foram feitas as configurações de obtenção dos dados e geração dos gráficos no servidor Zabbix. O resultado do monitoramento desse agente



usando a interface Web do Zabbix é mostrado na Figura 4.9. Com o agente funcionando, foram feitas as coletas do dados.

Figura 4.9 – Página do Zabbix monitorando dispositivo com agente SNMP.



Fonte: próprio autor.

## 4.7.2 Agente e protocolo Zabbix

Assim como feito com o protocolo anterior, o primeiro passo foi procurar biblioteca para o protocolo Zabbix. Como não havia biblioteca disponível, uma nova biblioteca foi implementada. A nova biblioteca foi chamada de ZabbixPassiveWifi, por tratar apenas o modo passivo desse protocolo, conforme explicado na seção 2.10.2, e por funcionar apenas em conexões de rede sem fio. O funcionamento apenas em modo passivo foi uma decisão de projeto, pois nesse modo a comunicação é bem simplificada, sendo trocados apenas texto plano entre o servidor e o agente, simplificando também a implementação. Já o fato de funcionar especificamente em comunicação Wi-fi se dá por restrições da API usada.

A biblioteca encapsula toda complexidade do protocolo, com o objetivo de deixar a implementação do agente o mais simples possível. Ela tem como principais funções:

- Receber e guardar um ponteiro para a função do agente que será invocada

quando forem recebidas requisições do servidor Zabbix;

- Abrir a porta TCP 10050 e ficar escutando requisições do Zabbix;
- Ao receber uma nova requisição do Zabbix, invocar a função de tratamento indicada pelo agente;
- Receber a resposta do agente para cada requisição;
- Construir a resposta à requisição com base no cabeçalho do protocolo e na resposta do agente;
- Devolver resposta à requisição do Zabbix.

Conforme descrito na seção 2.10.2, a identificação do item de gerenciamento é chamada pelo protocolo de chave. As chaves de monitoramento implementadas no agente são listadas na Tabela 4.3. Foram criadas chaves para o tratamento das seis informações previstas na seção 4.4. Adicionalmente, foi criada a chave “agent.ping” para tratar esse tipo de requisição muito comum no gerenciamento com o protocolo Zabbix.

Tabela 4.3 – Chaves implementadas no agente de monitoramento.

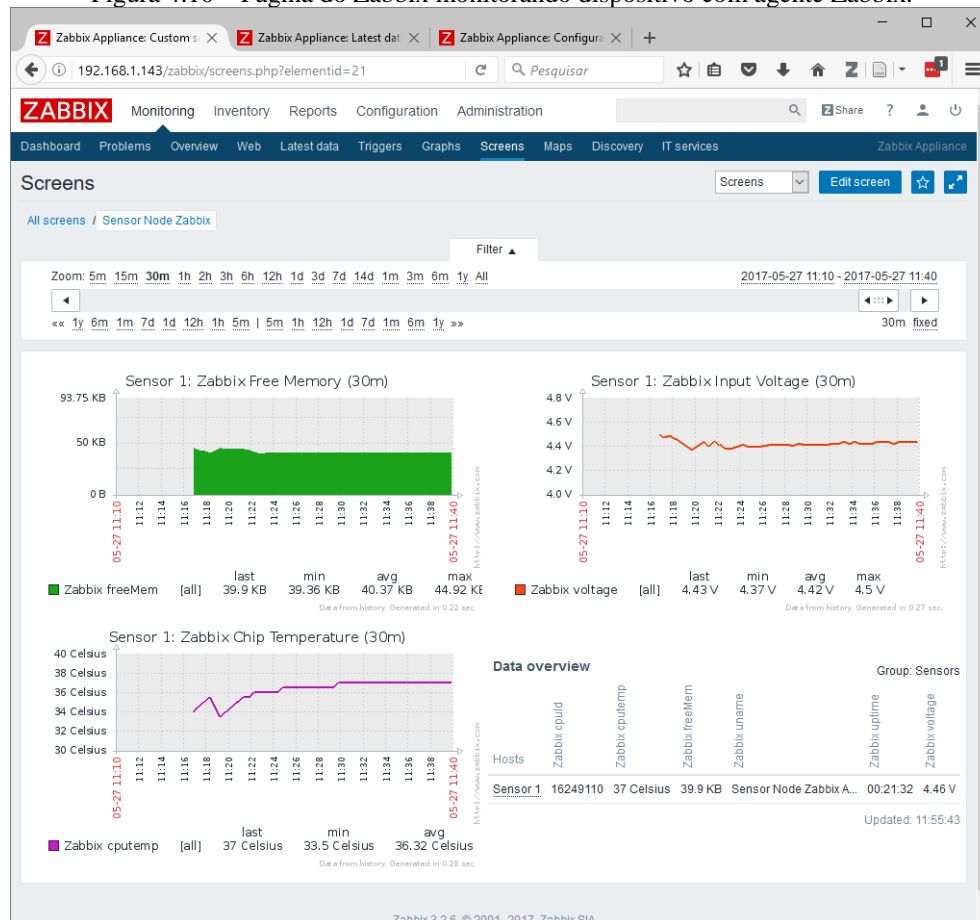
Chave	Retorno	Descrição
<b>agent.ping</b>	1: agente ativo Nada: agente inativo	Verifica se o agente está ativo.
<b>system.uname</b>	String	Retorna o nome do agente.
<b>system.uptime</b>	long	Retorna o tempo total de funcionamento do dispositivo (em segundos).
<b>system.stat[cpu,id]</b>	Integer	Retorna a identificação do microcontrolador.
<b>system.stat[memory,fre]</b>	long	Retorna a memória disponível no dispositivo (em bytes)
<b>node.cputemp</b>	float	Retorna a temperatura do microcontrolador (em graus Celsius).
<b>node.voltage</b>	float	Retorna a tensão de alimentação do dispositivo (em Volts).

Fonte: autoria própria.

A implementação do agente fez uso da biblioteca ZabbixPassiveWifi e, portanto, se resumiu a implementar as funções `setupZabbixAgent()` e `loopZabbixAgent()`, e a função de *call-back* `keyReceived()`. A função `setupZabbixAgent()` é chamada apenas no início de funcionamento do dispositivo e apenas registra na biblioteca a função de *call-back*. A função `loopZabbixAgent()` é chamada continuamente pelo dispositivo e sua implementação manda a biblioteca continuar escutando a porta para verificar novas conexões. A função de *call-back* é invocada pela biblioteca quando o dispositivo recebe alguma requisição do

servidor de gerenciamento. A função avalia a chave solicitada e responde com a informação correspondente à resposta para essa chave. O código fonte do agente e da biblioteca ZabbixPassiveWifi pode ser acessado no GitHub, conforme detalhado nos Apêndices A.1 e A.2.

Figura 4.10 – Página do Zabbix monitorando dispositivo com agente Zabbix.



Fonte: próprio autor.

Após a implementação do agente, foram feitas as devidas configurações no servidor Zabbix para acesso ao dispositivo. Entretanto, as primeiras medições mostraram um resultado bem diferente do esperado. Por ser um protocolo simplificado, acreditava-se que o consumo de memória ficaria bem abaixo dos outros protocolos, mas isso não ocorreu de início. Foi iniciado um processo de refatoramento para melhoria do código, principalmente em pontos que poderiam gerar economia de memória. Diversos atributos e variáveis, que inicialmente foram definidos como classes de objetos, foram substituídos por ponteiros. A biblioteca foi praticamente toda escrita novamente. Como resultado desse processo, houve redução principalmente de memória RAM estática.

A execução do agente Zabbix e o resultado do seu monitoramento é mostrado na Figura 4.10. Com o dispositivo funcionando durante um tempo, foram feitas as coletas das

métricas desejadas.

### 4.7.3 Agente e protocolo MQTT

Como o protocolo MQTT está em evidência desde 2015, foi possível encontrar uma biblioteca que já implementasse esse protocolo para a plataforma do experimento. A biblioteca PubSubClient<sup>30</sup>, desenvolvida por Nick O'Leary, já provê o tratamento necessário para um cliente MQTT assinar e publicar mensagens em um servidor MQTT. Em maio de 2017, momento da implementação do experimento, a biblioteca tinha algumas limitações, mas elas não afetaram o funcionamento do agente proposto nesse trabalho. Essas limitações são descritas a seguir:

- Publicação de mensagens apenas com um nível de serviço (QoS);
- Tamanho máximo da mensagem de 128 bytes, incluindo o cabeçalho;
- Tempo de manutenção da conexão ativa fixado em 15 segundos;
- Implementação da versão 3.1.1 do protocolo MQTT, não suportando versões anteriores.

O aproveitamento da biblioteca PubSubClient facilitou muito a implementação do agente. Mas seu funcionamento difere um pouco do funcionamento dos protocolos SNMP e Zabbix. No MQTT o agente funciona como um cliente, abrindo a conexão com Broker e publicando as informações por sua própria iniciativa, nos momentos e intervalos escolhidos por ele. O agente deve, em sua inicialização, abrir uma conexão com o Broker e, periodicamente, publicar as informações de monitoramento. As informações serão lidas por interessados que assinarem os respectivos tópicos.

Ao contrário dos protocolos anteriores, o funcionamento do MQTT exigiu algumas definições relacionadas ao agente:

- O agente precisa conhecer o endereço do Broker;
- A periodicidade de envio das informações é definida no agente;

Além disso, o modelo de funcionamento do mqttnwarn acrescentou mais um requisito:

---

<sup>30</sup> Disponível em <https://github.com/knolleary/pubsubclient>. Acesso em fevereiro de 2017.

- O agente deve publicar as informações em tópicos cujos nomes seguem a regra: “zabbix/item/sensor-node<último segmento do ip>/<item>”.

Tabela 4.4 – Tópicos implementados no agente de MQTT.

<b>Tópico (raiz: zabbix/item/sensor-node&lt;ip&gt;/)</b>	<b>Retorno</b>	<b>Descrição</b>
<b>.../system.uname</b>	<i>String</i>	Retorna o nome do agente.
<b>.../system.uptime</b>	<i>long</i>	Retorna o tempo total de funcionamento do dispositivo (em segundos).
<b>.../system.stat[cpu,id]</b>	<i>Integer</i>	Retorna a identificação do microcontrolador.
<b>.../system.stat[memory,fre]</b>	<i>long</i>	Retorna a memória disponível no dispositivo (em bytes)
<b>.../node.cputemp</b>	<i>float</i>	Retorna a temperatura do microcontrolador (em graus Celsius).
<b>.../node.voltage</b>	<i>float</i>	Retorna a tensão de alimentação do dispositivo (em Volts).

Fonte: autoria própria.

Como regra, o mqttwarn se conecta ao Broker e assina o tópico “zabbix/item/#”. Isso indica que qualquer publicação para um tópico cujo nome inicie com “zabbix/item/” será encaminhada para o Zabbix. O encaminhamento segue a regra indicada no caminho do tópico: “zabbix/item/<host>/<item>”. Esse encaminhamento chegará no Zabbix como uma mensagem TRAP, indicando que se trata do item de monitoramento <item>, relativo ao host <host>. O Zabbix captura a informação e armazena em seu banco de dados. A Figura 4.11 mostra alguns exemplos de publicações de um agente MQTT que está sendo executado em um dispositivo com IP 192.168.1.196. A Tabela 4.4 lista os tópicos implementados no agente, correspondentes aos itens de monitoramento.

Figura 4.11 – Publicações do agente MQTT.

```
zabbix/item/sensor-node196/system.uname Sensor Node MQTT Agent
zabbix/item/sensor-node196/system.uptime 725
zabbix/item/sensor-node196/system.stat[memory,fre] 44360
zabbix/item/sensor-node196/node.voltage 4.32
```

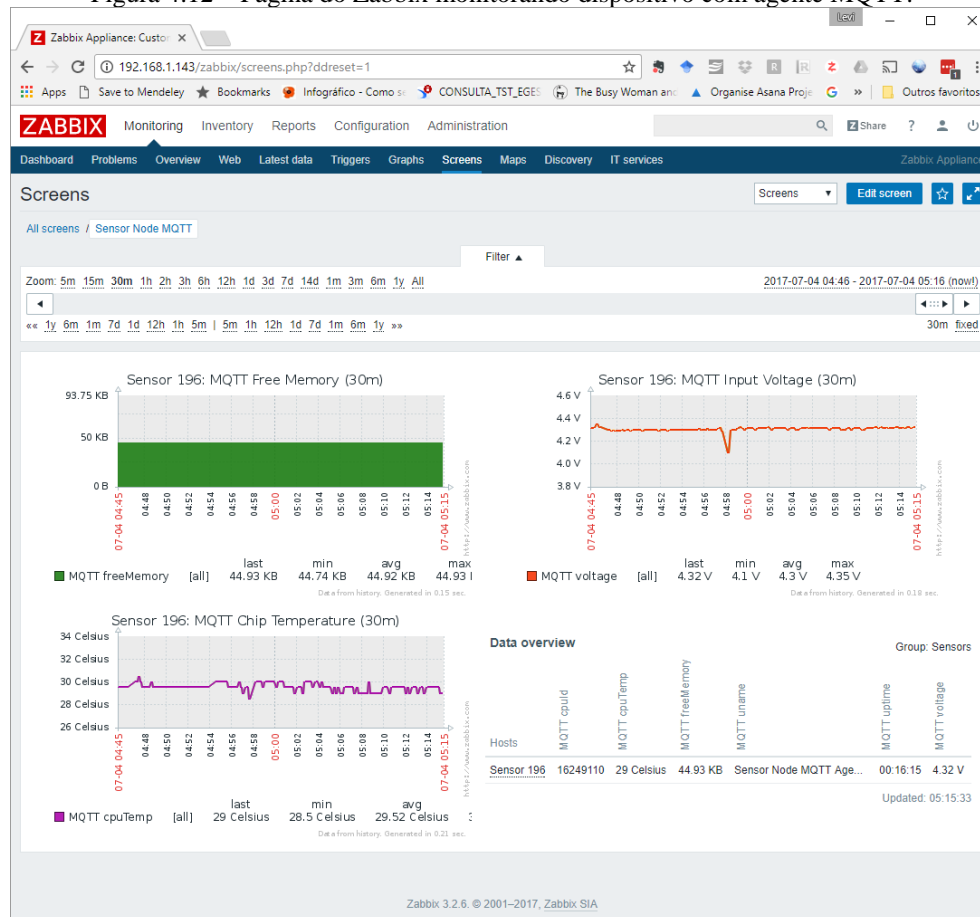
Fonte: próprio autor.

Com o funcionamento definido, o agente foi implementado com apenas três funções: `setupMqttAgent()`, `loopMqttAgent()` e `publishItem()`. A função `setupMqttAgent()` apenas estabelece a conexão com o servidor. A função `loopMqttAgent()`, que é chamada continuamente pelo dispositivo, verifica se já decorreu o tempo definido como periodicidade de envio, e invoca a função `publishItem()` para cada item, passando o nome e o dado de monitoramento. A função `publishItem()` monta o nome do tópico com base no nome do item

e pede à biblioteca que publique o tópico no Broker. O código fonte do agente está disponível no GitHub<sup>31</sup>, e seu acesso é detalhado nos Apêndices A.1 e A.2.

Assim como nos outros protocolos, o passo seguinte foi a configuração do Zabbix para receber as informações de monitoramento. Mas, para fazer a coleta desse agente, ainda foi necessário iniciar o Mosquitto e realizar a configuração e execução do mqttwarn. As configurações do mqttwarn foram feitas alterando os arquivos mqttwarn.ini e saplefuncs.py. Essas alterações são encontradas no APÊNDICE B.

Figura 4.12 – Página do Zabbix monitorando dispositivo com agente MQTT.



Fonte: próprio autor.

Com todos os componentes ativos, o monitoramento foi feito, como mostra a Figura 4.12, e a coleta das métricas de avaliação do protocolo foi realizada. Entretanto, ao realizar a coleta da métrica de energia consumida, percebeu-se que o modo de funcionamento ativo do protocolo MQTT poderia ser aproveitado para alterar o comportamento do dispositivo e reduzir o consumo de energia. Com o agente em modo ativo, a comunicação é iniciada no agente e não no servidor, como acontece nos outros protocolos analisados. Isso permite que a transmissão seja feita no momento mais adequado ao agente. Tal abordagem possibilita

<sup>31</sup> Disponível em <https://github.com/levicm/nes-management/>

que o dispositivo entre em modo de suspensão enquanto estiver entre os intervalos de coleta e envio do dado, reduzindo assim o consumo de energia e aumentando, em proporção inversa, a autonomia do dispositivo em casos de alimentação por bateria. O código do agente foi então alterado para suportar a suspensão do ESP8266 entre as coletas.

Para funcionar em modo de suspensão, foram acrescentadas as lógicas abaixo descritas:

- 1) SensorNode: inclusão da constante `SLEEP_MODE` por meio da diretiva `#define`, permitindo habilitar o comportamento em todo o firmware;
- 2) Wifi: dependendo da constante `SLEEP_MODE`, armazena na EEPROM do dispositivo as últimas configurações de IP recebidas do servidor DHCP da rede. Na inicialização do Wifi, verifica se está vindo de uma suspensão e, em caso positivo, usa as últimas configurações de IP armazenadas. Isso reduz o tempo de inicialização de três segundos para aproximadamente duzentos milissegundos, pois não há mais a necessidade de solicitar e aguardar o servidor DHCP;
- 3) MQTTAgent: dependendo da constante `SLEEP_MODE`, força a entrada em modo de suspensão após o envio das informações de monitoramento;
- 4) SystemInfo: dependendo da constante `SLEEP_MODE`, inicia o objeto Uptime em modo de persistência;
- 5) Biblioteca Uptime: implementação de modo de persistência que permite guardar o último tempo de funcionamento na memória do RTC para continuar a contagem após o retorno da suspensão.

## 4.8 Considerações finais do capítulo

Nesse capítulo foram detalhados todos os passos para o desenvolvimento do ambiente experimental. Foram descritas as atividades de projeto e construção do dispositivo e dos programas que continham os agentes para cada protocolo avaliado.

O Zabbix foi instalado e configurado a partir de sua distribuição em formato de máquina virtual tipo *appliance*. Isso facilitou todo o processo para a colocação em funcionamento do servidor e da interface de gerenciamento.

O dispositivo sensor foi desenvolvido para funcionar como um Mote. Foi usado o NodeMCU como base para o Mote construído, acrescentando componentes para darem a capacidade de sensoramento do ambiente e o fornecimento de informações relacionadas à falha e ao desempenho do dispositivo.

Foi implementado um agente de gerenciamento para cada protocolo analisado. A implementação do agente SNMP fez uso da biblioteca AgentuinoWifi, adaptada a partir da biblioteca livre Agentuino. O agente MQTT usou a biblioteca livre PubSubClient. E o agente Zabbix usou a biblioteca ZabbixPassiveWifi, totalmente implementada no escopo deste trabalho.

O objetivo foi ter agentes completamente funcionais, enviando dados de gerenciamento a serem monitorados pelo Zabbix, a fim de realizar as medições das métricas definidas. A análise dos resultados obtidos é feita no próximo capítulo.



## 5. Resultados

---

Neste capítulo são analisados os resultados da pesquisa experimental, a partir das medições coletadas durante a execução dos experimentos. Os dados são tabulados e gráficos são gerados para embasar a análise.

### 5.1 Cenário do experimento

Este trabalho usa como referência o cenário descrito por TELESCA *et al.* (2014) para avaliação do Zabbix como ferramenta usada no sistema de monitoramento do centro de dados do experimento ALICE (*A Large Ion Collider Experiment*), realizado no CERN. O sistema de aquisição de dados – ALICE *Data-Acquisition* (DAQ) coleta dados como utilização de CPU, utilização de memória e tráfego de rede. As coletas acontecem em intervalos de 10 a 60 segundos, sendo o pior caso, o intervalo de coleta de 10 em 10 segundos. E no caso de teste mais simples são coletadas de 1 a 10 informações por requisição (TELESCA *et al.*, 2014).

Com base no cenário usado por TELESCA *et al.* (2014) foi definido o cenário do experimento deste trabalho. Ele leva em conta a existência de um dispositivo sensor coletando as seguintes informações do ambiente:

- **Temperatura do ambiente**, em graus Celsius;
- **Umidade relativa do ar**, em porcentagem;

O dispositivo sensor é gerenciado e monitorado por um servidor Zabbix, que realiza, a cada dez segundos, a coleta dos dados relativos a falha e desempenho, conforme descrito a seguir:

- **Identificação do chip**: permite diferenciar os vários dispositivos espalhados em um ambiente de rede;
- **Tempo total de funcionamento**: permite saber o período de tempo que o dispositivo está funcionando sem interrupção;

- **Memória RAM disponível:** permite acompanhar se o dispositivo está trabalhando com memória disponível suficiente para o seu bom funcionamento. Também permite avaliar tendências de ocupação de memória, possibilitando ações proativas em dispositivos que poderão ficar sem memória disponível dentro de algum tempo;
- **Temperatura do chip:** permite identificar se a temperatura do chip está dentro da faixa aceitável pelo fabricante;
- **Tensão de alimentação do dispositivo:** permite acompanhar a tensão da fonte de alimentação e identificar se a carga da bateria está próxima do final. Se combinada com a medição da corrente drenada pelo dispositivo, permitiria acompanhar o nível de carga da fonte de alimentação e a autonomia do dispositivo;

Por conveniência, cada execução do experimento dura um tempo total de cinco minutos. Nesse período são feitas cento e cinquenta requisições do servidor para o agente. Pois, como o servidor dispara requisições a cada dez segundos, e nesse momento são solicitadas cinco informações de falha e desempenho diferentes, no período de cada prova tem-se um total de trinta rajadas de cinco requisições. Esse tempo é suficiente para confirmação do consumo dinâmico de memória dos agentes, cuja estabilização já acontece por volta dos três minutos de execução. Além disso, durante esse tempo é possível realizar 16.100 amostras de consumo de energia do dispositivo, quantidade suficiente para observar a linearidade na evolução deste consumo.

## 5.2 Consumo geral de memória

O resultado das coletas de memória ROM e memória RAM estática dos três agentes são mostrados na Tabela 5.1. Para a análise é preciso levar em conta que o ESP8266 analisado possui 512K bytes de memória ROM, que correspondem a 524.288 bytes, e 80K bytes de memória RAM disponíveis para o programa, que representam 81.920 bytes. Com base nisso, e analisando os resultados coletados, é possível fazer algumas observações.

Como a coleta foi feita por camadas, pode-se observar que a camada do SDK ocupa 221.995 bytes de memória ROM, correspondente a 42,3% do total, e 31.568 bytes de memória RAM, correspondente a 38,5% do total. A soma da memória ROM das demais

camadas é 7,7%, 9,5% e 12,3% da memória ROM ocupada pelo SDK nos protocolos SNMP, Zabbix e MQTT respectivamente. Para a memória RAM, essa comparação é de 5,6%, 4,8% e 6,6%, nos mesmos protocolos.

Tabela 5.1 – Uso de memória ROM e RAM dos agentes (em bytes).

Componente	SNMP		Zabbix		MQTT	
	ROM	RAM	ROM	RAM	ROM	RAM
<b>SDK (SO)</b>	221.995	31.568	221.995	31.568	221.995	31.568
<b>Rede e enlace</b>	1.018	76	1.018	76	1.018	76
<b>UDP</b>	2.256	128	0	0	0	0
<b>TCP</b>	0	0	3.208	192	3.208	192
<b>Firmware básico</b>	9.392	1.016	9.392	1.016	9.392	1.016
<b>Agente</b>	4.408	552	7.416	240	13.668	812
<b>Total <i>firmware</i></b>	<b>239.069</b>	<b>33.340</b>	<b>243.029</b>	<b>33.092</b>	<b>249.281</b>	<b>33.664</b>

Fonte: próprio autor.

### 5.3 Memória ROM

Os resultados da ocupação de memória ROM pelos agentes são mostrados na Tabela 5.2 e na Figura 5.1. Nesse aspecto o protocolo SNMP foi melhor e ocupou no total 239.069 bytes, 3.960 bytes a menos que o *firmware* do Zabbix com total de 243.029, e 10.212 bytes a menos que o *firmware* do MQTT com total de 249.281.

Tabela 5.2 – Uso de memória ROM dos agentes (em bytes).

Componente	SNMP	Zabbix	MQTT
<b>SDK (SO)</b>	221.995	221.995	221.995
<b>Rede e enlace</b>	1.018	1.018	1.018
<b>UDP</b>	2.256	0	0
<b>TCP</b>	0	3.208	3.208
<b>Firmware básico</b>	9.392	9.392	9.392
<b>Agente</b>	4.408	7.416	13.668
<b>Total</b>	<b>239.069</b>	<b>243.029</b>	<b>249.281</b>

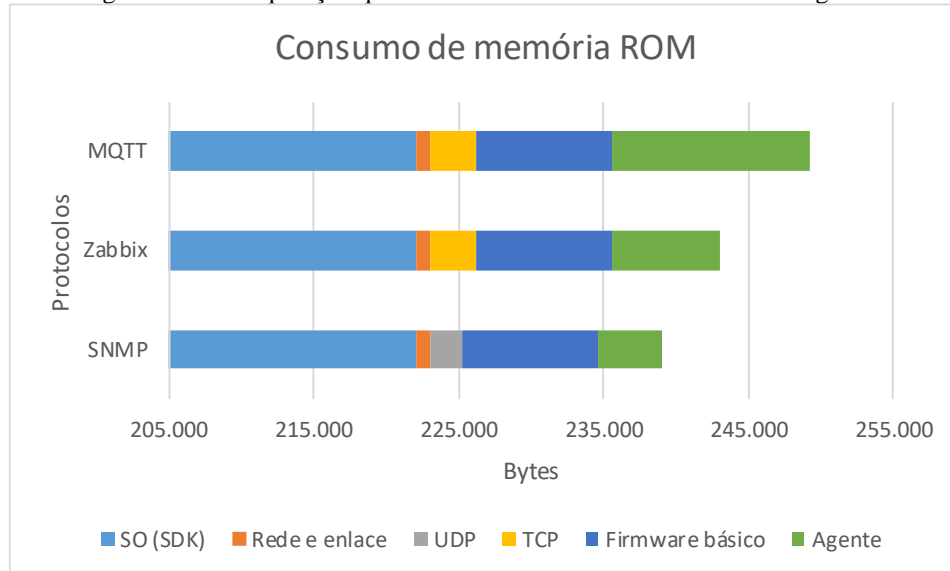
Fonte: próprio autor.

Pode-se observar que as camadas do SDK, rede e enlace, e o *firmware* básico se mantém iguais entre os protocolos. O gráfico mostrado na Figura 5.1 dá uma ideia visual da diferença entre eles. É possível perceber que a diferença ficou por conta do protocolo de transporte, usado pelos agentes, e por conta do código do próprio agente.

Em relação ao protocolo de transporte, o SNMP usa mensagens UDP e isso se mostrou uma vantagem em termos de ocupação de memória ROM quando comparado com o Zabbix e o MQTT, que usam conexões TCP. O protocolo UDP usou 2.256 bytes,

enquanto o protocolo TCP usou 3.208 bytes, equivalente a 42,2% a mais que o UDP. Isso se deve ao fato do protocolo TCP ser orientado a conexão (confiável), enquanto o UDP não tem essa preocupação.

Figura 5.1 – Comparação quanto ao uso de memória ROM entre os agentes.



Fonte: próprio autor.

O espaço ocupado pelo *firmware* básico foi de 9.392 bytes. Esse espaço foi maior que o espaço ocupado apenas pelos agentes SNMP e Zabbix. Isso pode ser atribuído ao fato de que essa camada contém todo o código para obtenção dos dados dos sensores e o código responsável por obter as informações de falha e desempenho. Isso engloba a biblioteca para o DHT22, as bibliotecas OneWire e DallasTemperature para o DS18B20, e a biblioteca Uptime para obtenção do tempo total de funcionamento.

Analisando apenas o espaço ocupado pelo próprio agente, o SNMP ocupou 4.408 bytes, enquanto o Zabbix ocupou 7.416 bytes, 68% a mais, e o MQTT ocupou 13.668 bytes, 84% maior que o Zabbix e 210% maior que o SNMP. A menor ocupação de memória ficou por conta do agente SNMP. Esse desempenho é creditado à maturidade da biblioteca usada. A base da biblioteca AgentuinoWifi foi a biblioteca Agentuino, publicada no Google Code por Eric Gionet em 2010. Entre 2010 a 2015 a biblioteca Agentuino teve uma grande evolução, resultado da correção de problemas e aplicação de melhorias reportadas por vários usuários.

O agente Zabbix ficou em segundo lugar em ocupação de memória ROM. Isso aconteceu mesmo com o trabalho de refatoramento feito na biblioteca Zabbix, visto que o processo de afinação do código dessa biblioteca focou na redução de ocupação de memória

RAM, não se preocupando com o espaço ocupado em memória ROM.

A maior ocupação de memória ROM foi do agente MQTT. Isso se deve ao fato desse agente usar a biblioteca PubSubClient, que é bem sofisticada e isso resulta em mais código. Essa biblioteca tem diversos tratamentos para permitir atender às funcionalidades do MQTT, como assinar e publicar tópicos. Ela usa muitos objetos em seus atributos e parâmetros de métodos. As classes disponibilizam vários métodos para a mesma funcionalidade, dando diversas opções ao utilizador. Tudo isso traz impacto na ocupação de memória ROM pelo código resultante.

## 5.4 Memória RAM inicial (estática)

Os resultados da ocupação de memória RAM estática são mostrados na Tabela 5.3 e na Figura 5.2. Tomando como base o total de memória RAM ocupada, o protocolo Zabbix se saiu melhor, ocupando no total 33.092 bytes, 248 bytes a menos que o *firmware* do SNMP com total de 33.340, e 572 bytes a menos que o *firmware* do MQTT com total de 33.664. Observando as camadas individualmente, fica claro que o responsável por isso foi o próprio agente do Zabbix, que conseguiu ocupar apenas 240 bytes, contra 552 bytes do agente SNMP e 812 bytes do agente MQTT.

Tabela 5.3 – Uso de memória RAM dos agentes (em bytes).

Componente	SNMP	Zabbix	MQTT
SDK (SO)	31.568	31.568	31.568
Rede e enlace	76	76	76
UDP	128	0	0
TCP	0	192	192
Firmware básico	1.016	1.016	1.016
Agente	552	240	812
Total	33.340	33.092	33.664

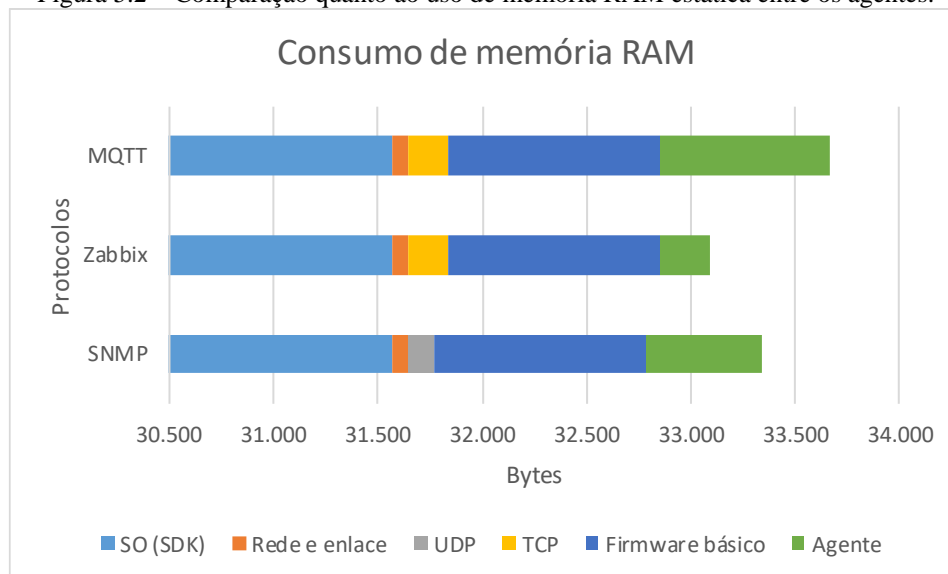
Fonte: próprio autor.

O resultado melhor do agente Zabbix pode ser atribuído ao trabalho de melhoria do código feito em tempo de desenvolvimento do experimento. As melhorias focaram em substituir a definição de variáveis *String* por vetores de caracteres, além do uso de ponteiros e passagem de parâmetros por referência.

Assim como na avaliação de memória ROM, a maior ocupação de memória RAM foi do agente MQTT. A mesma análise feita para a memória ROM pode ser aplicada aqui, já que a biblioteca PubSubClient, usada pelo agente, é bem sofisticada, o que resulta em maior

ocupação de memória RAM estática.

Figura 5.2 – Comparação quanto ao uso de memória RAM estática entre os agentes.



Fonte: próprio autor.

O comportamento das demais camadas é semelhante ao da memória ROM. As camadas do SDK e de rede e enlace são iguais entre os protocolos. O protocolo SNMP usa o protocolo UDP na camada de transporte, resultando numa ocupação menor de memória RAM do que a ocupação do protocolo TCP, que é usado tanto pelo Zabbix quanto pelo MQTT. O espaço ocupado pelo *firmware* básico foi de 1.016 bytes. Esse espaço foi maior que o espaço ocupado apenas pelos agentes SNMP e Zabbix. E, assim como na memória ROM, o motivo pode ser por essa camada conter todo o código para obtenção dos dados dos sensores do ambiente e informações de falha e desempenho, juntamente com todas as bibliotecas envolvidas.

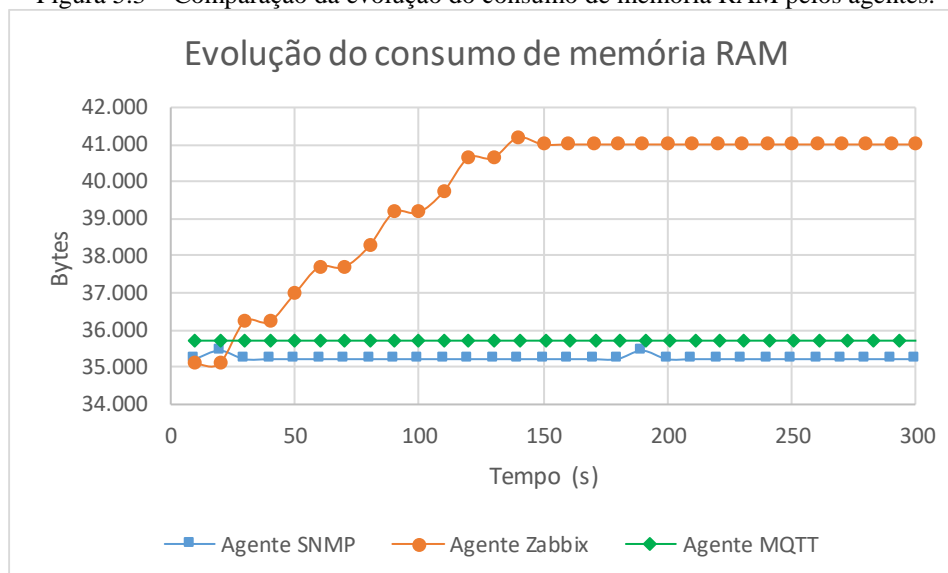
Chama a atenção o fato do *firmware* básico ocupar mais memória RAM estática do que os agentes. Possivelmente isso é resultado da concentração de código e de variáveis nessa camada. É no *firmware* básico que se encontra a rotina de conexão com a rede, a classe que encapsula a obtenção dos sensores de ambiente e a classe que encapsula a obtenção de informações sobre falha e desempenho do dispositivo.

## 5.5 Evolução do consumo total de memória RAM

Os resultados da coleta de memória RAM ocupada pelo *firmware* ao longo do tempo podem ser vistos, de forma gráfica, na Figura 5.3. A avaliação da evolução do consumo de memória RAM usou como base o cenário descrito na seção 5.1, com requisições do servidor

Zabbix acontecendo a cada dez segundos. Os dispositivos tiveram seu funcionamento monitorado por cinco minutos. Nesse período são tratadas requisições para medição de cinco itens de monitoramento. Durante o período de avaliação foram coletadas amostras da memória livre do dispositivo e em seguida tabuladas e analisadas. O detalhamento das coletas pode ser visto no APÊNDICE C.

Figura 5.3 – Comparação da evolução do consumo de memória RAM pelos agentes.



Fonte: próprio autor.

A ocupação total de memória RAM dos agentes parte, inicialmente, de patamares próximos. O agente SNMP inicia ocupando 35.224 bytes, o agente Zabbix inicia em 35.112 e o agente inicia MQTT em 35.728 bytes. A ocupação inicial tem relação direta com a ocupação de dados estáticos. Como já descrito nas seções 2.11.1 e 3.4.1, quando o dispositivo é ligado, a memória RAM é ocupada inicialmente pelo código que será executado e pelos dados estáticos. Em seguida, variáveis começam a ser alocadas no *heap* e as chamadas de funções vão alocando espaço na pilha, resultando em ocupação do restante da memória RAM.

A ocupação de memória RAM do agente SNMP se mantém, durante toda execução, no patamar de 35.224 bytes, salvo apenas dois momentos em que essa ocupação sobe rapidamente para 35.424 bytes. Isso mostra que as mensagens UDP que são trocadas ao longo do tempo pouco afetam a ocupação de memória RAM.

Já o agente Zabbix, que inicia ocupando 35.112 bytes de memória RAM, sendo um pouco menos que o SNMP. Todavia, ele sofre aumentos de memória à medida que vai recebendo requisições do servidor. As requisições do servidor Zabbix para esse protocolo são feitas sobre conexões TCP. No momento do estabelecimento de cada conexão TCP há

um incremento de aproximadamente 200 bytes de memória RAM, que só são liberados algum tempo depois de finalizada a conexão. Como a cada dez segundos são realizadas cinco novas requisições, nesses momentos são incrementados mais de 1Kb no uso de memória RAM. As conexões anteriores vão sendo descartadas e esse processo se repete. O intervalo de dez segundos entre as requisições do servidor é menor que o tempo de descarte das conexões, fazendo com que a memória siga aumentando. Ao atingir o patamar de 41.024 bytes tem-se um ponto de equilíbrio. Nesse momento, a alocação de memória pelo início de novas conexões e o descarte de memória das conexões anteriores mantém a ocupação de memória no mesmo patamar até o fim da execução do experimento.

Já o protocolo MQTT inicia a execução ocupando 35.728 bytes e se mantém nesse patamar até o final da execução do experimento. Isso pode ser explicado porque, pela sua característica, conforme descrito na seção 2.10.3, o protocolo MQTT usa apenas uma conexão do tipo TCP. De maneira que todas as trocas de mensagens acontecem sobre a mesma conexão. Isso ocupa um espaço de memória da ordem de 200 bytes, mas se mantém estável durante toda a execução do agente.

## **5.6 Consumo de energia elétrica**

Conforme descrito na seção 3.4.2, para se chegar no consumo de energia elétrica, também são medidas a tensão elétrica e a corrente elétrica. O produto dessas duas últimas grandezas resulta na potência dissipada. O consumo de energia é a integral das potências ao longo do tempo.

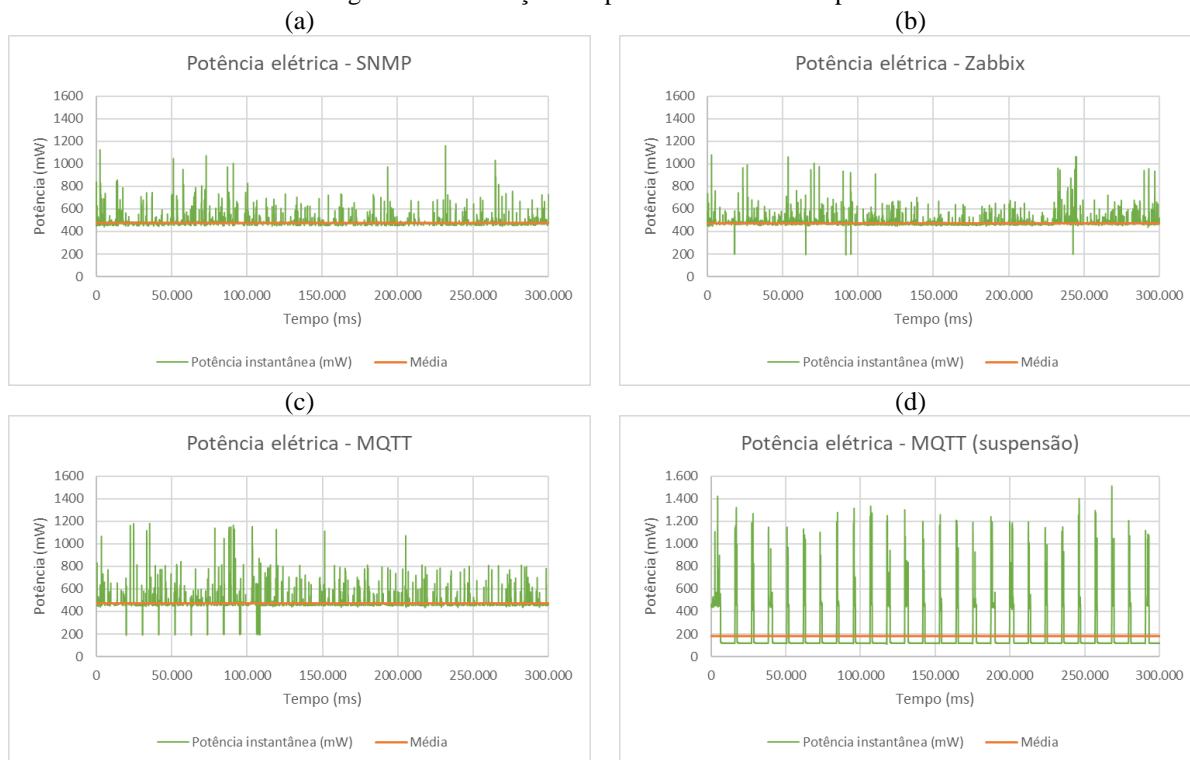
Os gráficos mostrando as medições de potência ao longo do tempo de execução do experimento são apresentados na Figura 5.4. Foram incluídas medições realizadas com os agentes de cada um dos protocolos analisados. Apesar de serem observados picos de potência que chegam a superar os 1200 mW, a potência média dissipada se mantém na faixa dos 475 mW independente do protocolo usado. Essa análise também pode ser verificada na Tabela 5.4.

O destaque fica por conta do experimento realizado com o protocolo MQTT que coloca o dispositivo em modo de suspensão nos intervalos entre as coletas e envios dos dados. Conforme pode ser observado na Figura 5.4 (d), o dispositivo inicia seu funcionamento dissipando uma potência média de 475mW, porém, após o primeiro envio de informações e o dispositivo ser colocado em modo de suspensão profunda, a potência



dissipada passa para um patamar de 120mW. Quando o dispositivo retorna do modo de suspensão, a potência tem picos de mais de 1200mW voltando para o patamar de 475mW e iniciando novo ciclo. Os resultados detalhados das medições de tensão, corrente, potência, energia e previsão de descarga da bateria de todos os agentes são trazidos no APÊNDICE D.

Figura 5.4 – Medições de potência elétrica dissipada.



Fonte: próprio autor.

Tabela 5.4 – Consumo de energia dos agentes.

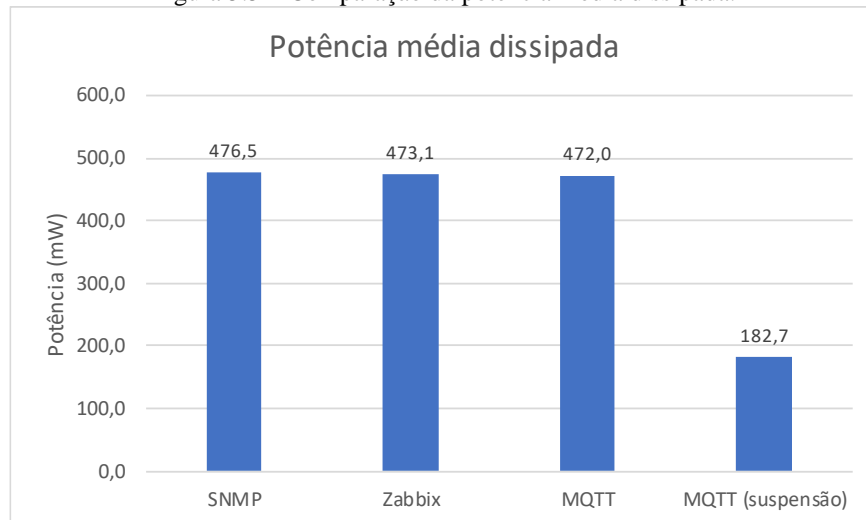
<b>Firmware e agente</b>	<b>Média da potência dissipada (mW)</b>	<b>Energia total consumida (mWh)</b>	<b>Previsão de duração de bateria de 4300mAh (horas)</b>
Agente SNMP	476,5	39,7	43,4
Agente Zabbix	473,1	39,6	43,8
Agente MQTT	475,0	39,6	43,9
Agente MQTT (com modo de suspensão)	182,7	15,1	115,1

Fonte: próprio autor.

A Tabela 5.4 mostra os resultados das medições da média da potência dissipada, como também detalha a energia total consumida nos cinco minutos de execução dos experimentos. Na tabela também é incluída uma previsão de duração de bateria 4300mAh se considerarmos a carga consumida durante o experimento. Pode-se observar novamente que o grande diferencial acontece quando o dispositivo é colocado em modo de suspensão. Nesse caso, a média da potência dissipada durante toda a execução do experimento é de 182,7mW, ante aos 475mW dos demais casos. A energia total consumida no experimento é de 15,1mWh, contra a média de 39,6 dos demais casos. E a duração prevista da bateria passa

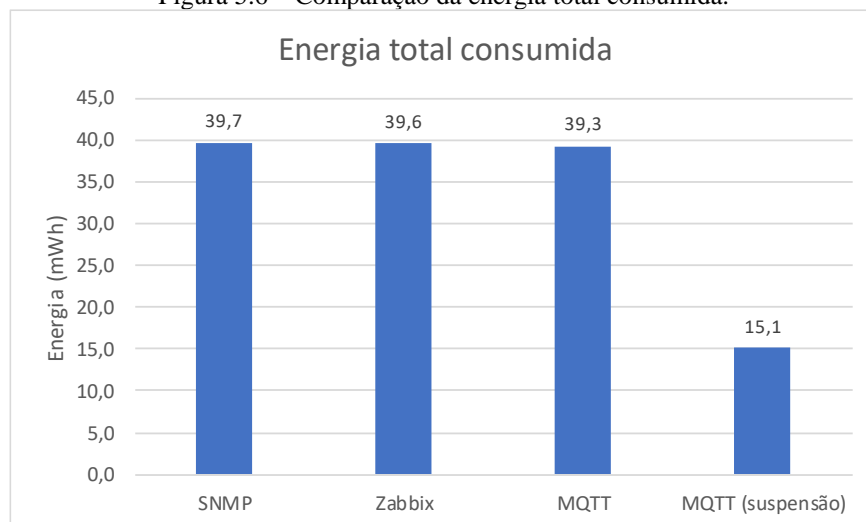
de 43,7 horas para mais de 115 horas.

Figura 5.5 – Comparação da potência média dissipada.



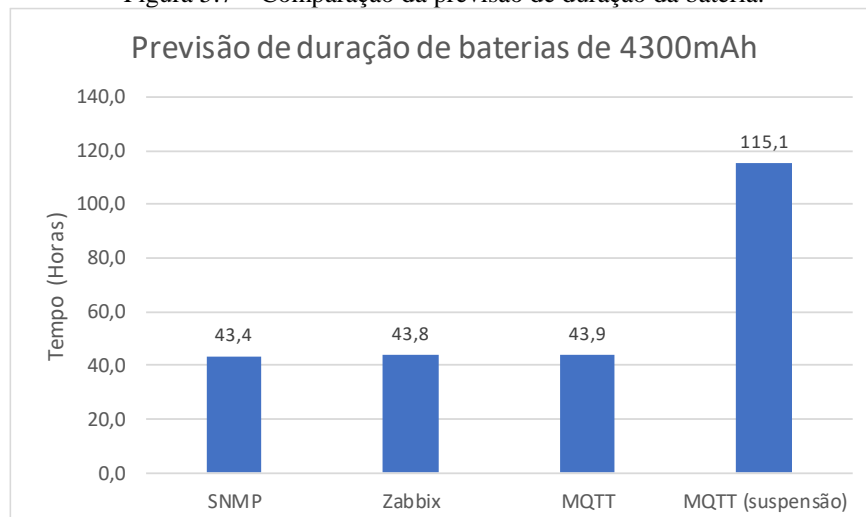
Fonte: próprio autor.

Figura 5.6 – Comparação da energia total consumida.



Fonte: próprio autor.

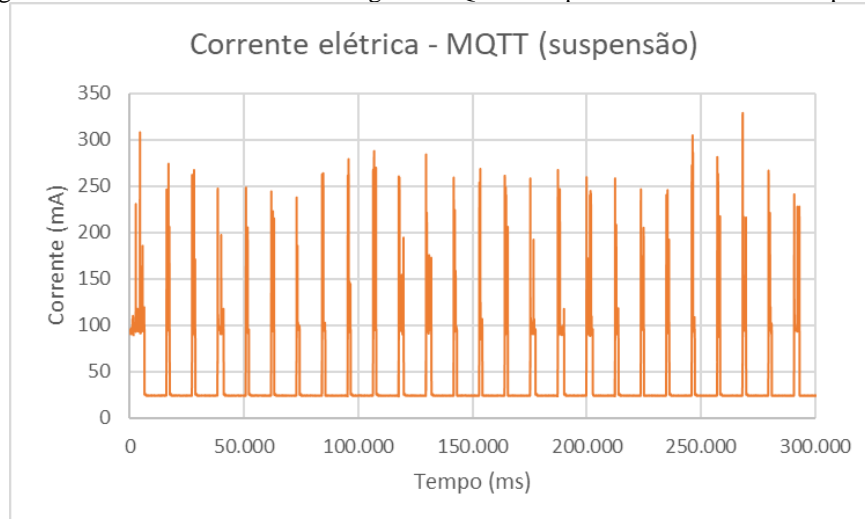
Figura 5.7 – Comparação da previsão de duração da bateria.



Fonte: próprio autor.

A Figura 5.5 mostra um gráfico comparativo das médias de potência dissipada nos casos analisados. O gráfico que compara as energias totais consumidas por cada caso pode ser visto na Figura 5.6. E a Figura 5.7 traz um gráfico que faz a comparação entre as previsões de descarga de bateria de 4300mAh.

Figura 5.8 – Corrente medida com agente MQTT e dispositivo entrando em suspensão.



Fonte: próprio autor.

Ressalta-se que, apesar do consumo de energia com o uso do protocolo MQTT e a colocação do dispositivo em modo de suspensão ter sido 37% do consumo dos demais casos, ele ainda foi bem superior ao esperado para esse caso. A Figura 5.8 mostra a corrente elétrica medida nessa situação. É possível observar que mesmo em modo de suspensão ainda é verificada uma corrente de mais de 20 mA. Isso é mil vezes maior do que consta nas especificações do ESP8266 detalhadas na seção 2.11, que seria de 20  $\mu$ A para o modo de suspensão profundo, que foi usado no trabalho. A suspeita aqui é de que a placa NodeMCU esteja acrescentando esse consumo adicional, pois ela inclui regulador de tensão e circuito de comunicação USB.

## 6. Conclusões

---

Este trabalho realizou uma análise comparativa de protocolos de gerenciamento de redes aplicados a um ambiente de IoT. Um ambiente experimental completo foi montado para permitir o monitoramento de dispositivos sensores em uma rede TCP/IP. Para funcionar como dispositivo sensor, foi construído um Mote sobre o ESP8266, acrescentando componentes sensores para obtenção de temperatura e umidade do ambiente, e incluindo funcionalidades para coleta de informações de falha e desempenho do dispositivo. Os protocolos SNMP, Zabbix e MQTT foram analisados sob o aspecto de ocupação de memória RAM e sob o aspecto do consumo de energia elétrica do dispositivo gerenciado.

O problema, descrito na seção 1.1, refere-se à necessidade de conhecer as melhores alternativas de protocolos, entre os analisados, para uso no gerenciamento de elementos de rede aplicados à Internet das Coisas, no que tange ao uso de memória e ao consumo de energia adicionados ao dispositivo pelo agente. Após a execução do experimento e coleta das medições, foi possível responder à Questão de Pesquisa, descrita na seção 1.2, na medida em que o consumo de recursos, tanto de memória quanto de energia, acrescentado pelos agentes dos protocolos analisados, foram apresentadas na seção 5. Além disso, como desdobramento da análise dos resultados obtidos, foi possível obter mais algumas conclusões:

- a) Os três protocolos analisados podem ser implementados e executados em dispositivos ESP8266;
- b) O agente do protocolo SNMP foi o que se saiu melhor em termos de ocupação de memória, tanto ROM quanto RAM, e é a melhor alternativa, nesse aspecto, entre os três protocolos. Em muitos cenários, o uso de um protocolo padronizado e consolidado como SNMP pode ser desejável, ou até requerido. Nesses casos, o SNMP atende perfeitamente;
- c) O agente do protocolo Zabbix teve o segundo melhor desempenho em ocupação de memória ROM. Em relação à memória RAM estática, ele foi melhor que os

outros dois protocolos. Entretanto, quando analisada a evolução da ocupação memória RAM ao longo do tempo do experimento, este protocolo foi o pior entre os três. É possível inferir que esse protocolo não resista a execuções em cenários de maior sobrecarga de requisições do que o cenário analisado, que teve seis solicitações a cada dez segundos durante cinco minutos;

- d) O agente do protocolo MQTT, apesar ter sido o que mais ocupou memória ROM e memória RAM estática, se manteve estável durante toda a execução e se mostra uma alternativa viável caso o ambiente de aplicação exija ou tenha preferência por esse recente protocolo;
- e) Não há diferença significativa de consumo de energia entre os protocolos analisados na plataforma ESP8266 com placa de desenvolvimento NodeMCU;
- f) O agente do protocolo MQTT, por seu funcionamento, permitiu que o dispositivo fosse colocado em modo de suspensão reduzindo em mais da metade o consumo de energia nesse modelo;
- g) Com base no consumo de energia observado e na previsão de duração da bateria, é inviável o uso do ESP8266 com a placa NodeMCU em aplicações de campo que façam uso de bateria e que exijam longa autonomia, mesmo no caso do protocolo MQTT com dispositivo entrando em modo de suspensão.

Ao final deste trabalho, e como resultado da pesquisa experimental realizada aqui, é possível elencar as contribuições a seguir:

- a) Foi desenvolvido um método para experimentação e análise de plataformas ESP8266 aplicadas à Internet das Coisas, que pode ser usado por pesquisadores que desejam dar continuidade a pesquisas semelhantes nesse ambiente;
- b) Foram obtidos resultados que mostram as possibilidades de uso dos protocolos analisados no ambiente de Internet das Coisas;
- c) Foi desenvolvido e construído um Mote, plenamente funcional, com base no ESP8266 e alguns componentes adicionais;
- d) A biblioteca de suporte ao SNMP encontrada, Agentuino, foi portada para uso em conexões de rede sem fio e disponibilizada para a comunidade, com nome de

AgentuinoWifi;

- e) Foi desenvolvida uma biblioteca para dar suporte ao protocolo Zabbix em modo passivo, aplicável à plataforma ESP8266, com nome de ZabbixPassiveWifi;
- f) Foi implementado um agente de gerenciamento SNMP para o ESP8266, usando a biblioteca AgentuinoWifi;
- g) Foi implementado um agente de gerenciamento Zabbix para o ESP8266, usando a biblioteca ZabbixPassiveWifi;
- h) Foi implementado um agente de gerenciamento MQTT para o ESP8266, usando a biblioteca de terceiros PubSubClient;
- i) Foi desenvolvido um dispositivo para medição de grandezas elétricas relacionadas ao consumo de energia elétrica em corrente contínua de outros equipamentos.

## 6.1 Trabalhos futuros

Como possíveis trabalhos futuros, buscando a continuidade da pesquisa e o aperfeiçoamento da proposta deste trabalho, destacam-se:

Realização de experimento semelhante, incluindo mais protocolos na análise, como o Modbus, por exemplo.

Aplicação de experimento semelhante, realizando a análise do tráfego de rede dos agentes de cada protocolo. Pois quanto menor o tráfego, menor é o esforço do dispositivo em enviar e receber dados.

Aplicação de experimento semelhante, realizando a análise da latência na troca das mensagens entre o dispositivo e o servidor.

Realização de experimento semelhante, avaliando o consumo de energia entre os protocolos com base no funcionamento da placa ESP8266 isolada de qualquer outra plataforma de desenvolvimento, como a NodeMCU.

Analisar a possibilidade de aplicação aos outros protocolos, além do MQTT, o modelo de gerenciamento de agente ativo, avaliando seus ganhos. Nesse modelo, a comunicação é iniciada no agente e não no servidor, como é mais comum. Isso permite que a transmissão

seja feita no momento mais adequado ao agente. Tal abordagem possibilita que o dispositivo entre em modo de suspensão enquanto estiver entre os intervalos de coleta e envio do dado, reduzindo assim o consumo de energia e aumentando, em proporção inversa, a autonomia do dispositivo em casos de alimentação por bateria.

Propor alterações no agente MQTT que permitam que este seja inicialmente configurado pelo servidor, já que, pela arquitetura do protocolo, neste experimento o dispositivo precisou ter previamente o endereço do Broker que receberia as publicações de informações. Essa configuração inicial passada pelo servidor, poderia indicar ao dispositivo qual o endereço do servidor e o intervalo desejado, evitando que essas informações sejam fixadas no agente.

Realização de estudo sobre os requisitos de segurança da solução, como confidencialidade, integridade e disponibilidade.

# Referências

AI-THINKER. Datasheet: ESP-12E WiFi Module. *Product Datasheet*, p. 1–18, 2015.

AL-FUQAHA, A. *et al.* Internet of Things : A Survey on Enabling Technologies, Protocols, and Applications. *IEEE COMMUNICATION SURVEYS & TUTORIALS*, v. 17, n. 4, p. 2347–2376, 2015.

ATZORI, L.; IERA, A.; MORABITO, G. The Internet of Things: A survey. *Computer Networks*, v. 54, n. 15, p. 2787–2805, 2010. Disponível em: <<http://linkinghub.elsevier.com/retrieve/pii/S1389128610001568>>.

BLANCHET, B. An efficient cryptographic protocol verifier based on prolog rules. *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.*, p. 82–96, 2001.

CAI, X. *et al.* Design and implementation of a WiFi sensor device management system. *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, CoAP Avalia memória ocupada Tamano dos pacotes, p. 10–14, 2014.

CASE, J. D. *et al.* Simple network management protocol (SNMP). *IETF*, p. 1–36, 1990.

CHOI, H.; KIM, N.; CHA, H. 6LoWPAN-SNMP: Simple Network Management Protocol for 6LoWPAN. *2009 11th IEEE International Conference on High Performance Computing and Communications*, p. 305–313, 2009. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5167008>>.

CROSSBOW. Mica2 Datasheet. *Product Datasheet*, p. 2, 2005. Disponível em: <<http://www.eol.ucar.edu/isf/facilities/isa/internal/CrossBow/DataSheets/mica2.pdf>>.

CROSSBOW. MICA2DOT datasheet. *Product Datasheet*, p. 1–2, 2002.

CROSSBOW TECHNOLOGY. MICAz: Wireless Measurement System. *Product Datasheet*, p. 4–5, 2008. Disponível em: <[http://courses.ece.ubc.ca/494/files/MICAz\\_Datasheet.pdf](http://courses.ece.ubc.ca/494/files/MICAz_Datasheet.pdf)>.

DAGALE, H. *et al.* CyPhyS+: A Reliable and Managed Cyber-Physical System for Old-Age Home Healthcare over a 6LoWPAN Using Wearable Motes. *2015 IEEE International Conference on Services Computing*, p. 309–316, 2015. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7207368>>.

DINI, G.; SAVINO, I. M. A Security Architecture for Reconfigurable Networked Embedded Systems. *International Journal of Wireless Information Networks*, v. 17, n. 1–2, p. 11–25, 2010. Disponível em: <<http://link.springer.com/10.1007/s10776-010-0116-y>>.

ESPRESSIF SYSTEMS. ESP8266EX Datasheet Version 5.4. *Product Datasheet*, p. 1–31, 2017. Disponível em: <[http://espressif.com/sites/default/files/documentation/0a-esp8266ex\\_datasheet\\_en.pdf](http://espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf)>.

ESTRIN, D. *et al.* Connecting the physical world with pervasive networks. *IEEE Pervasive Computing*, v. 1, n. 1, p. 59–69, 2002. Disponível em:



<<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=993145>>.

ETIENNE, S.-. Evaluation of SNMP-like Protocol to Manage a NoC Emulation Platform. 2014.

FENG, K.; HUANG, X.; SU, Z. A network management architecture for 6LoWPAN network. *4th IEEE International Conference on Broadband Network and Multimedia Technology, IC-BNMT 2011*, p. 430–434, 2011.

FERNANDES, A. C. *et al.* Sistema de aquisição de sinais ECG processado pelo LabVIEW com comunicação wi-fi por meio do módulo ESP8266. *Revista Principia*, v. 34, p. 62–68, 2017. Disponível em: <<http://periodicos.ifpb.edu.br/index.php/principia/article/download/1337/667>>.

GEORGESCU, M.; HUCANU, R. An Approach About Turning Challenges Into Opportunities Using Internet Of Things. *The 12 th International Scientific Conference eLearning and Software - for Education*, p. 12753, 2016.

GERHARDT, T. E.; SILVEIRA, D. T. *Métodos de Pesquisa*. Porto Alegre: Editora UFRGS, 2009.

HILL, J.; CULLER, D. A wireless embedded sensor architecture for system-level optimization. p. 12, 2001. Disponível em: <[http://webs.cs.berkeley.edu/papers/MICA\\_ARCH.pdf](http://webs.cs.berkeley.edu/papers/MICA_ARCH.pdf)>.

HILL, J. L.; CULLER, D. E. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, v. 22, n. 6, p. 12–24, 2002.

HORST, A. H. S.; PIRES, A. S.; DÉO, A. L. B. *De A a Zabbix*. 1. ed. São Paulo: Novatec Editora Ltda., 2015.

KAKANAKOV, N.; KOSTADINOVA, E. Using SNMP for Remote Measurement and Automation. 2007.

KERRISON, S.; EDER, K. Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor. *ACM Transactions on Embedded Computing Systems*, v. 14, n. 3, p. 56:1-56:25, 2015. Disponível em: <<http://doi.acm.org/10.1145/2700104>>.

KODALI, R. K.; SORATKAL, S. MQTT based home automation system using ESP8266. *2016 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*, n. May, p. 1–5, 2016. Disponível em: <<http://ieeexplore.ieee.org/document/7906845/>>.

KOLBAN, N. Kolban's Book on ESP8266. p. 436, 2016.

KURYLA, S.; SCHÖNWÄLDER, J. Evaluation of the resource requirements of SNMP agents on constrained devices. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 6734 LNCS, p. 100–111, 2011.

LAMPKIN, V. *et al.* Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ Telemetry. *IBM Redbooks*, p. 270, 2012. Disponível em: <[http://books.google.com/books?hl=en&lr=&id=F\\_HHAgAAQBAJ](http://books.google.com/books?hl=en&lr=&id=F_HHAgAAQBAJ)>.

- LEVIS, P. *et al.* TinyOS: An Operating System for Wireless Sensor Networks. *Ambient Intelligence*, p. 115–148, 2005. Disponível em: <<http://www.springerlink.com/index/10.1007/b138670>>.
- LIGHT, R. A. Mosquitto : server and client implementation of the MQTT protocol. n. May, p. 10–11, 2017.
- LU, Y. F. *et al.* A perl-based SNMP agent of networked embedded devices for smart-living applications. 2015, Denpasar, Indonesia: IEEE, 2015. p. 342–347.
- MAGHETI, M. A.; CIOBANU, A. N.; POPOVICI, E. C. Network Management Extensions , Performing Network Management Activities. Fala das funções de gerenciamento, p. 173–176, 2010.
- MARQUES, G.; PITARMA, R. An indoor monitoring system for ambient assisted living based on internet of things architecture. *International Journal of Environmental Research and Public Health*, v. 13, n. 11, 2016.
- MCROBERTS, M. Arduino Básico. p. 456, 2011.
- MOUI, A.; DESPRATS, T. Towards Self-Adaptive Monitoring Framework for Integrated Management. *Managing the Dynamics of Networks and Services*, p. 160–163, 2011.
- MUKHTAR, H. *et al.* LNMP- Management architecture for IPv6 based low-power wireless Personal Area Networks (6LoWPAN). *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, Fala das fuões do gerenciamentoUsa 6LoWPANAvalia apenas nr de hops, p. 417–424, 2008.
- OLIVEIRA, R. R. USO DO MICROCONTROLADOR ESP8266 PARA AUTOMAÇÃO RESIDENCIAL. p. 55, 2017.
- PAVENTHAN, A. *et al.* WSN monitoring for agriculture: Comparing SNMP and emerging CoAP approaches. abr. 2013, [S.l.]: IEEE, abr. 2013. p. 353–358. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6757167>>.
- RANTOS, K. *et al.* Secure policy-based management solutions in heterogeneous embedded systems networks. *2012 International Conference on Telecommunications and Multimedia (TEMU)*, p. 227–232, 2012. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6294723>>.
- RAZZAQUE, M. A. *et al.* Middleware for Internet of Things: A Survey. *IEEE Internet of Things Journal*, v. 3, p. 70–95, 2016. Disponível em: <<http://tarjomefa.com/wp-content/uploads/2016/10/5464-Eenglish.pdf>>.
- ROSE, M. T.; MCCLOGHRIE, K. Management Information Base for network management of TCP/IP-based internets: MIB-II. *IETF*, p. 1–70, 1991. Disponível em: <<https://www.rfc-editor.org/pdf/rfc/rfc1213.txt.pdf>>.
- SANTOS, W. S. E. *et al.* Miniestação agronomica baseada na plataforma ESP8266 para aplicações agrícolas. 2016.
- SEHGAL, A. *et al.* Management of resource constrained devices in the internet of things. *Communications Magazine, IEEE*, v. 50, n. 12, p. 144–149, 2012. Disponível em:

<[http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6384464&matchBoolean=true&rowsPerPage=30&searchField=Search\\_All&queryText=\(p\\_DOI:10.1109/MCOM.2012.6384464\)>](http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6384464&matchBoolean=true&rowsPerPage=30&searchField=Search_All&queryText=(p_DOI:10.1109/MCOM.2012.6384464)>).

SHELBY, Z.; BORMANN, C. *6LoWPAN: the wireless embedded internet*. [S.l.]: John Wiley & Sons Ltd, 2009.

SHENG, Z. *et al.* Lightweight Management of Resource Constrained Sensor Devices in Internet-of-Things. *IEEE Internet of Things Journal*, v. PP, n. 99, p. 1–1, 2015. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7080876>>.

SHENG, Z. *et al.* Recent Advances in Industrial Wireless Sensor Networks Toward Efficient Management in IoT. *IEEE Access*, v. 3, n. Oma Dm, p. 622–637, 2015.

STEFANOV, I. FS20 / SNMP Gateway on TinyOS. p. 1–20, 2008.

TANG, C. BIN. Explore o MQTT e o serviço de Internet of Things no IBM Bluemix saber como funciona. *IBM Developer Works*, p. 1–19, 2015. Disponível em: <<https://www.ibm.com/developerworks/br/cloud/library/cl-mqtt-bluemix-iot-node-red-app/cl-mqtt-bluemix-iot-node-red-app-pdf.pdf>>.

TELESCA, A. *et al.* System performance monitoring of the ALICE Data Acquisition System with Zabbix. *Journal of Physics: Conference Series*, v. 513, p. 1–7, 2014.

UYTTERHOEVEN, P. *Zabbix Cookbook*. Birmingham: Packt Publishing Ltd, 2015. Disponível em: <<https://books.google.com.br/books?hl=pt-BR&lr=&id=gNqFBwAAQBAJ>>.

VASSEUR, J.-P.; DUNKELS, A. Interconnecting Smart Objects with IP. *Interconnecting Smart Objects with IP*. Burlington: Elsevier Ltd, 2010. p. 407. Disponível em: <<http://www.sciencedirect.com/science/article/pii/B9780123751652000223>>.

WANG, Q. *et al.* Reprogramming wireless sensor networks: challenges and approaches. *IEEE Network*, v. 20, n. 3, p. 48–55, 2006.

WOLF, W. What is embedded computing? *Computer*, v. 35, n. 1, p. 136–137, 2002. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=976929>>.

WU, X.; ZHU, Y.; DENG, X. Design and Implementation of Embedded SNMP Network Management Manager in Web-Based Mode. *Proc. IEEE Asia-Pacific Services Computing Conference APSCC '08*, p. 1512–1516, 2008.

ZANELLA, A. *et al.* Internet of Things for Smart Cities. *IEEE Internet of Things Journal*, v. 1, n. 1, p. 22–32, 2014. Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6740844>>.

ZURAWSKI, R. *Embedded Systems Handbook*. Boca Raton: CRC Press, 2005. Disponível em: <<https://books.google.com/books?hl=pt-BR&lr=&id=qYAqBgAAQBAJ>>. Acesso em: 22 jan. 2016.

## **Apêndices**

## APÊNDICE A Código fonte dos dispositivos

O código fonte e outros arquivos necessários para compilar os projetos estão disponíveis publicamente para download e experimentação no serviço GitHub.

### A.1 Fontes do *firmware* e dos agentes implementados para o dispositivo sensor

O acesso ao repositório do código fonte pode ser feito acessando a seguinte estrutura:

- **Repositório:** <https://github.com/levicm/nes-management/>
  - **SensorNode:** fontes do *firmware* do dispositivo
    - **Debug.h:** cabeçalho com macros para facilitar a depuração;
    - **MQTTAgent.ino:** agente para o protocolo MQTT;
    - **SensorNode.ino:** fonte principal do *firmware*;
    - **Sensors.h:** cabeçalho com classe que encapsula a obtenção dos sensores de ambiente;
    - **SnmpAgent.ino:** agente para o protocolo SNMP;
    - **SystemInfo.h:** cabeçalho que encapsula a obtenção de informações sobre falha e desempenho do dispositivo;
    - **Wifi.ino:** fonte com rotinas de rede;
    - **ZabbixAgent.ino:** agente para o protocolo Zabbix;

### A.2 Fontes das bibliotecas implementadas para o dispositivo sensor

- **Repositório:** <https://github.com/levicm/Arduino-libs/>
  - **AgentuinoWifi:** biblioteca SNMP com suporte a conexão Wifi;
  - **UptimeESP:** biblioteca que implementa contabilização de tempo total de funcionamento do dispositivo;
  - **Zabbix:** biblioteca que implementa o protocolo de agente Zabbix.

### A.3 Fontes do dispositivo medidor de energia

- **Repositório:** <https://github.com/levicm/EnergyMonitor/>

# APÊNDICE B Configurações feitas no mqttwarn

## B.1 Arquivo mqttwarn.ini

Arquivo com as configurações do mqttwarn que habilitam o funcionamento do plugin para o Zabbix e indica o arquivo samplefuncs.py como sendo o fonte Python que terá as funções que serão chamadas para o tratamento dos tópicos publicados.

```
[defaults]
hostname = 'localhost'
port     = 1883

; name the service providers you will be using.
launch   = file, log, zabbix
functions = 'samplefuncs'

[config:file]
append_newline = True
targets = {
    'mylog'      : [ '/temp/mqtt.log' ]
}

[config:log]
targets = {
    'info'      : [ 'info' ]
}

[test/+]
targets = file:mylog, log:info

[config:zabbix]
targets = {
    # Trapper address  port
    't1'      : [ '192.168.1.143', 10051 ],
}

[zabbix/clients/+]
alldata = ZabbixData()
targets = zabbix:t1, file:mylog, log:info

[zabbix/item/#]
alldata = ZabbixData()
targets = zabbix:t1, file:mylog, log:info
```

## B.2 Arquivo samplefuncs.py

Arquivo fonte Python que tem a função chamada para o tratamento dos tópicos publicados a serem redirecionados para o Zabbix.

```
# If the topic begins with zabbix/clients we have a host going up or down
# e.g. "zabbix/clients/jog03" -> "jog03"
#   extract client name (3rd part of topic)
#   set status key (e.g. 'host.up') to publish 1/0 on it (e.g during LWT)
#
# if the topic starts with zabbix/item we have an item/value for the host
# e.g. "zabbix/item/jog03/time.stamp" -> "jog03"
#   extract client name (3rd part of topic)
#
def ZabbixData(topic, data, srv=None):
    client = 'unknown'
    key = None
    status_key = None

    parts = topic.split('/')
    client = parts[2]

    if topic.startswith('zabbix/clients/'):
        status_key = 'host.up'

    if topic.startswith('zabbix/item/'):
        key = parts[3]

    return dict(client=client, key=key, status_key=status_key)
```

## APÊNDICE C Planilhas com as coletas de memória RAM em execução

Aqui são trazidas as coletas de memória RAM dos agentes ao longo da execução do experimento em cada um dos protocolos implementados.

### C.1 Agente SNMP

Amostra	Tempo (s)	Memória ocupada (bytes)	Memória livre (bytes)	Memória total (bytes)
1	10	35224	46696	81920
2	20	35424	46696	81920
3	30	35224	46696	81920
4	40	35224	46696	81920
5	50	35224	46696	81920
6	60	35224	46696	81920
7	70	35224	46696	81920
8	80	35224	46696	81920
9	90	35224	46696	81920
10	100	35224	46696	81920
11	110	35224	46696	81920
12	120	35224	46696	81920
13	130	35224	46696	81920
14	140	35224	46696	81920
15	150	35224	46696	81920
16	160	35224	46696	81920
17	170	35224	46696	81920
18	180	35224	46696	81920
19	190	35424	46696	81920
20	200	35224	46696	81920
21	210	35224	46696	81920
22	220	35224	46696	81920
23	230	35224	46696	81920
24	240	35224	46696	81920
25	250	35224	46696	81920
26	260	35224	46696	81920
27	270	35224	46696	81920
28	280	35224	46696	81920
29	290	35224	46696	81920
30	300	35224	46696	81920



## C.2 Agente Zabbix

Amostra	Tempo (s)	Memória ocupada (bytes)	Memória livre (bytes)	Memória total (bytes)
1	10	35112	46808	81920
2	20	35112	46808	81920
3	30	36240	45680	81920
4	40	36240	45680	81920
5	50	36976	44944	81920
6	60	37712	44208	81920
7	70	37712	44208	81920
8	80	38264	43656	81920
9	90	39184	42736	81920
10	100	39184	42736	81920
11	110	39736	42184	81920
12	120	40656	41264	81920
13	130	40656	41264	81920
14	140	41208	40712	81920
15	150	41024	40896	81920
16	160	41024	40896	81920
17	170	41024	40896	81920
18	180	41024	40896	81920
19	190	41024	40896	81920
20	200	41024	40896	81920
21	210	41024	40896	81920
22	220	41024	40896	81920
23	230	41024	40896	81920
24	240	41024	40896	81920
25	250	41024	40896	81920
26	260	41024	40896	81920
27	270	41024	40896	81920
28	280	41024	40896	81920
29	290	41024	40896	81920
30	300	41024	40896	81920

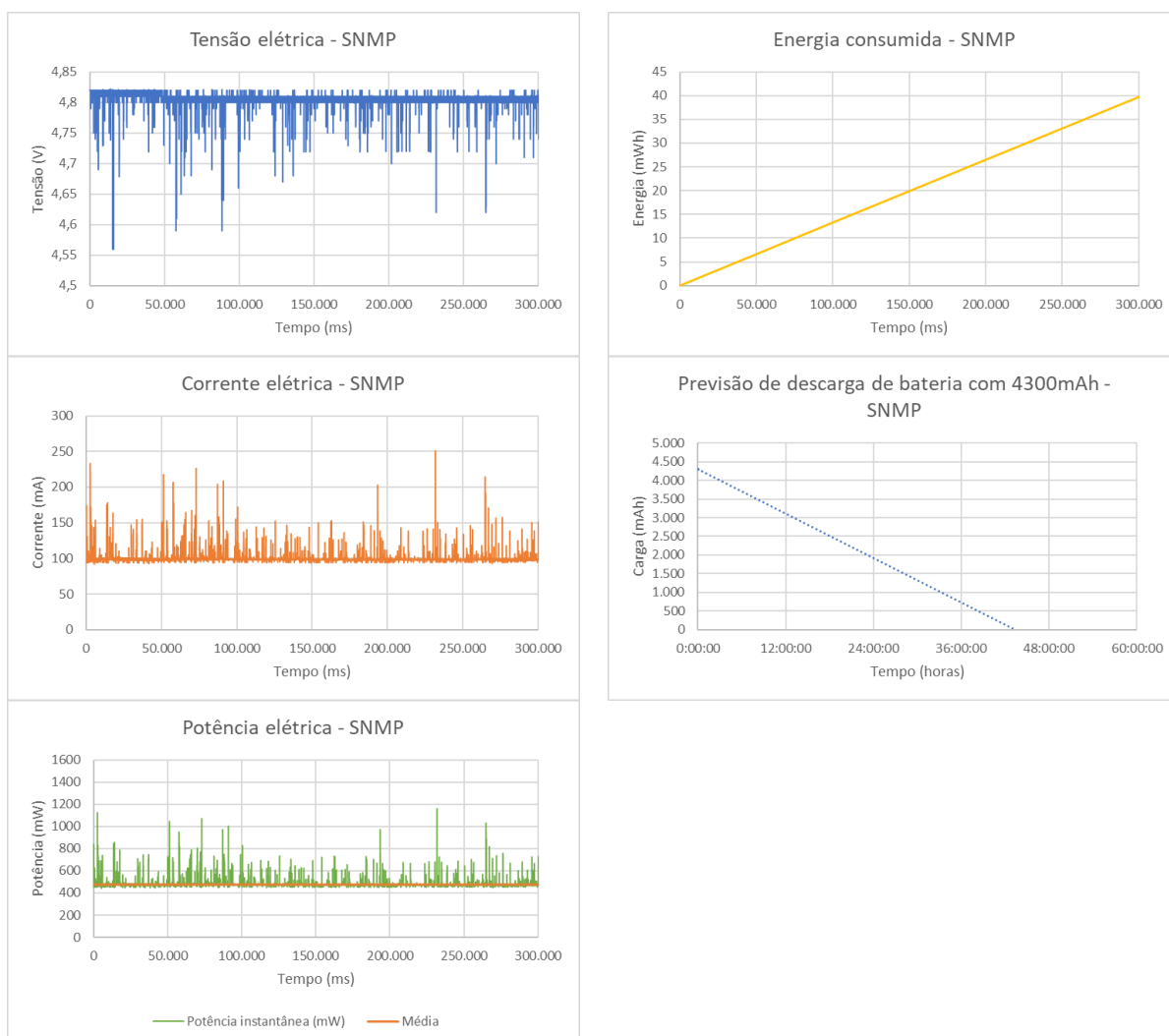
### C.3 Agente MQTT

Amostra	Tempo (s)	Memória ocupada (bytes)	Memória livre (bytes)	Memória total (bytes)
1	10	35728	46192	81920
2	20	35728	46192	81920
3	30	35728	46192	81920
4	40	35728	46192	81920
5	50	35728	46192	81920
6	60	35728	46192	81920
7	70	35728	46192	81920
8	80	35728	46192	81920
9	90	35728	46192	81920
10	100	35728	46192	81920
11	110	35728	46192	81920
12	120	35728	46192	81920
13	130	35728	46192	81920
14	140	35728	46192	81920
15	150	35728	46192	81920
16	160	35728	46192	81920
17	170	35728	46192	81920
18	180	35728	46192	81920
19	190	35728	46192	81920
20	200	35728	46192	81920
21	210	35728	46192	81920
22	220	35728	46192	81920
23	230	35728	46192	81920
24	240	35728	46192	81920
25	250	35728	46192	81920
26	260	35728	46192	81920
27	270	35728	46192	81920
28	280	35728	46192	81920
29	290	35728	46192	81920
30	300	35728	46192	81920

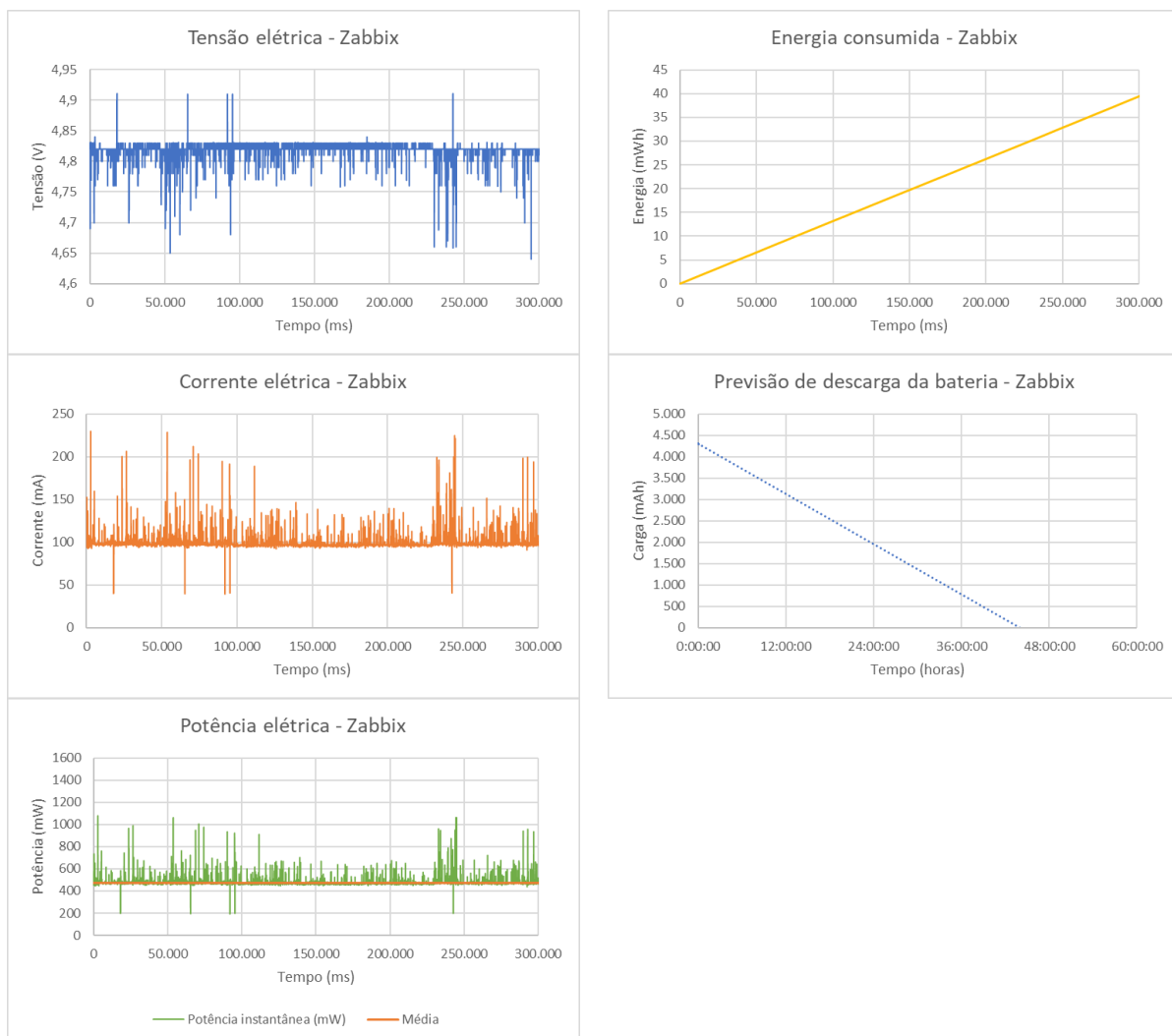
## APÊNDICE D Gráficos das grandezas elétricas medidas para cada protocolo

Aqui são trazidos os gráficos das grandezas elétricas medidas com o dispositivo em funcionamento para cada protocolo analisado.

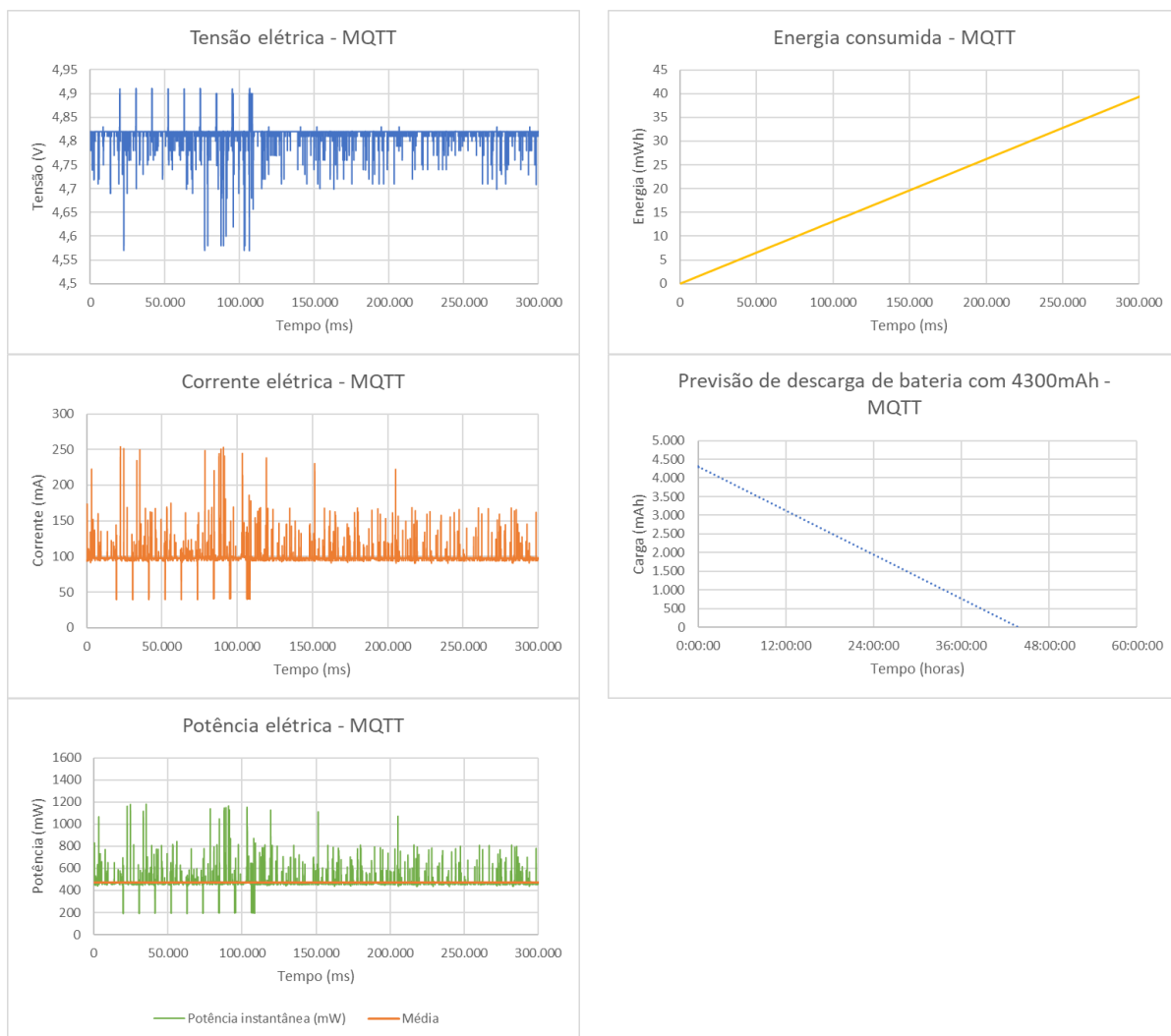
### D.1 Agente SNMP



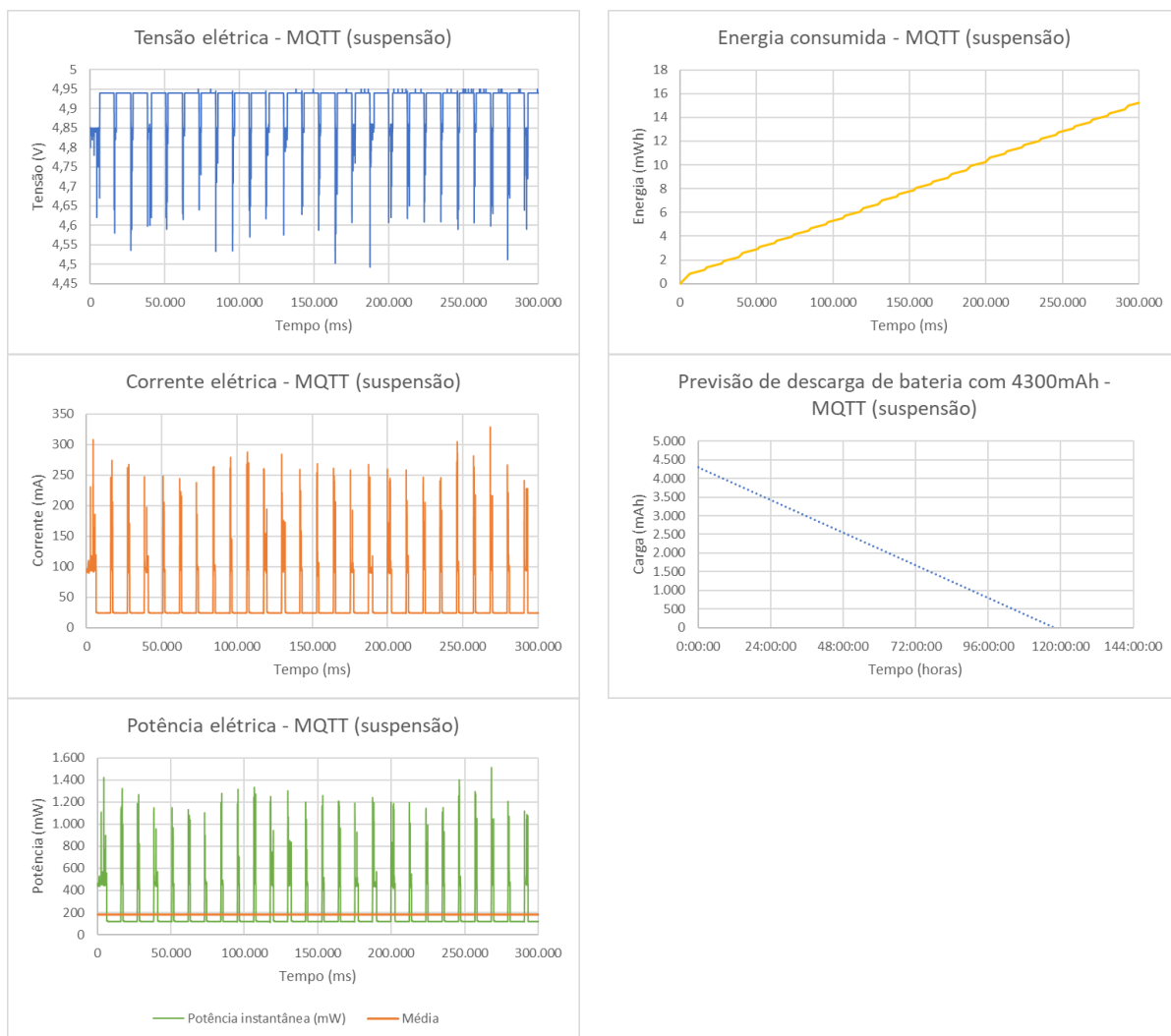
## D.2 Agente Zabbix



## D.3 Agente MQTT



## D.4 Agente MQTT funcionando em modo de suspensão



## **Anexos**

# ANEXO A    Fichas informativas (*datasheets*) de placas e componentes

Os datasheets das placas e componentes utilizados no experimento foram obtidos em endereços públicos da Internet, catalogados e incluídos no repositório do experimento. O acesso ao repositório do experimento pode ser feito usando o serviço GitHub e acessando a seguinte estrutura:

## A.1 Datasheets de componentes usados no dispositivo sensor

- **Repositório:** <https://github.com/levicm/nes-management/>
  - **docs/datasheets**
    - **am2302.pdf:** AM2302/DHT22 – Sensor digital de temperatura e umidade da Aosong;
    - **ds18b20.pdf:** DS18B20 – Sensor digital de temperatura da Dallas Semiconductor;
    - **esp-12e.pdf:** ESP-12E – Placa com ESP8266 da AI-Thinker;
    - **esp8266ex.pdf:** ESP8266 – SoC da Espressif;
    - **lm358.pdf:** LM358 – Amplificador Operacional da Motorola;

## A.2 Datasheets de componentes usados no dispositivo medidor de energia

- **Repositório:** <https://github.com/levicm/EnergyMonitor/>
  - **docs/datasheets**
    - **Arduino Nano-Rev3.2-SCH.pdf:** Arduino Nano;
    - **Datasheet\_INA219.pdf:** INA219 – Sensor de corrente elétrica baseado em resistor shunt, da Texas Instruments.



## ANEXO B Bibliotecas de terceiros usadas no experimento

Para a implementação do *firmware* usado no experimento foram necessárias algumas bibliotecas de terceiros listadas abaixo:

Nome	Uso	Desenvolvedor	Endereço
<b>Agentuino</b>	Suporte ao SNMP	Eric C. Gionet	<a href="https://code.google.com/archive/p/agentuino/">https://code.google.com/archive/p/agentuino/</a>
<b>Arduino-Temperature-Control-Library</b>	Sensor de temperatura DS18B20	Miles Burton	<a href="https://github.com/milesburton/Arduino-Temperature-Control-Library">https://github.com/milesburton/Arduino-Temperature-Control-Library</a>
<b>DHT-sensor-library</b>	Sensor de temperatura e umidade DHT22	Adafruit	<a href="https://github.com/adafruit/DHT-sensor-library">https://github.com/adafruit/DHT-sensor-library</a>
<b>PubSubClient</b>	Protocolo MQTT	Nick O'Leary	<a href="https://github.com/knolleary/pubsubclient">https://github.com/knolleary/pubsubclient</a>
<b>Adafruit_INA219</b>	Sensor de corrente elétrica INA219	Adafruit	<a href="https://github.com/adafruit/Adafruit_INA219">https://github.com/adafruit/Adafruit_INA219</a>